Evolutionary Scheduling of Dynamic Multitasking Workloads for Big-Data Analytics in Elastic Cloud

FAN ZHANG¹, (Senior Member, IEEE), JUNWEI CAO², (Senior Member, IEEE), WEI TAN³, (Senior Member, IEEE), SAMEE U. KHAN⁴, (Senior Member, IEEE), KEQIN LI⁵, (Senior Member, IEEE), AND ALBERT Y. ZOMAYA⁶, (Fellow, IEEE)

¹Kavli Institute for Astrophysics and Space Research, Massachusetts Institute of Technology, Cambridge, MA 02139 USA
²Research Institute of Information Technology, Tsinghua University, Beijing 100084, China
³IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 USA
⁴North Dakota State University, Fargo, ND 58102 USA
⁵Department of Computer Science, Tsinghua University, Beijing 100084, China
⁶School of Information Technologies, The University of Sydney, Sydney, NSW 2006, Australia

CORRESPONDING AUTHOR: F. ZHANG and J. CAO (zf19830622@gmail.com; jcao@mail.tsinghua.edu.cn)

This work was supported in part by Ministry of Science and Technology of China under National 973 Basic Research Program (grants No. 2013CB228206 and No. 2011CB302505), National Natural Science Foundation of China (grant No. 61472200 and No. 61233016) and National Science Foundation under grant CCF-1016966.

ABSTRACT Scheduling of dynamic and multitasking workloads for big-data analytics is a challenging issue, as it requires a significant amount of parameter sweeping and iterations. Therefore, real-time scheduling becomes essential to increase the throughput of many-task computing. The difficulty lies in obtaining a series of optimal yet responsive schedules. In dynamic scenarios, such as virtual clusters in cloud, scheduling must be processed fast enough to keep pace with the unpredictable fluctuations in the workloads to optimize the overall system performance. In this paper, ordinal optimization using rough models and fast simulation is introduced to obtain suboptimal solutions in a much shorter timeframe. While the scheduling solution for each period may not be the best, ordinal optimization can be processed fast in an iterative and evolutionary way to capture the details of big-data workload dynamism. Experimental results show that our evolutionary approach compared with existing methods, such as Monte Carlo and Blind Pick, can achieve higher overall average scheduling performance, such as throughput, in real-world applications with dynamic workloads. Furthermore, performance improvement is seen by implementing an optimal computing budget allocating method that smartly allocates computing cycles to the most promising schedules.

INDEX TERMS Big-data, cloud computing, evolutionary ordinal optimization, multitasking workload, virtual clusters.

I. INTRODUCTION

Large-scale business and scientific applications are usually composed of big-data, multitasking, time-variant, and fluctuating workloads [5], [35]. Cloud computing [2], with virtualization [3] as the key enabling technology, provides an elastic scaling-up and scaling-down provisioning mechanism. Agile and appropriate computational resource provisioning that keeps pace with the fluctuations of big-data multitasking workloads is very important in the scheduling paradigm [41]. For example, there are unprecedented amounts of requests from the customers of Amazon and eBay during the holiday seasons. Thereafter, the website traffic drops down dramatically. Over-provisioning of computing resource to constantly satisfy the requirements at the peak level leads to high and unnecessary cost, while under-provisioning leads to user churn [28].

In general, scheduling big-data multitasking workloads onto distributed computing resources is an NP-complete
Our major contribution consists of the following: usage of ordinal optimization to further reduce the scheduling workloads. Methods still refrains its applicability in highly fluctuated performance. However, the scheduling overhead of applying those load characteristics and lead to an overall improved performance. Experiments show that the methods capture workloads by reducing the exhaustive searching schedules instead of an optimal one results in much lower of the workloads. Targeting at a sequence of suboptimal solutions in much shorter time and with reduced runs and analysis of the workloads. Targeting at a sequence of suboptimal schedules instead of an optimal one results in much lower scheduling overhead by reducing the exhaustive searching time [40]. Experiments show that the methods capture workload fluctuations and resource provisioning in virtual clusters to achieve better performance.

In our previous work [40]–[43], for the purpose of fast scheduling, we have explored the possibility of applying simulation-based optimization methods. Ordinal Optimization (OO) [15] has been applied for suboptimal but fast searching. The OO is utilized for searching suboptimal solutions in much shorter time and with reduced runs and analysis of the workloads. Targeting at a sequence of suboptimal solutions instead of an optimal one results in much lower scheduling overhead by reducing the exhaustive searching time [40]. Experiments show that the methods capture workload fluctuations and resource provisioning in virtual clusters to achieve better performance.

In this paper, we propose an iterative and evolutionary usage of ordinal optimization to further reduce the scheduling overhead. Our major contribution consists of the following:

- The iterative OO (iOO) method published in our earlier paper [40], is applied in the new and real multitasking-scheduling model in this paper. As expected, the iOO method shows up to 20% performance speedup than competing methods. In this paper, we propose an iterative and evolutionary usage of ordinal optimization to further reduce the scheduling overhead. Our major contribution consists of the following:

- In a series of rapidly fluctuating workload periods, we contribute a procedure of using short-phase scheduling for fine-grained workload analysis. In stable workload phases, long-term scheduling is accordingly contributed which intends to save time to analyze the workloads for accurate scheduling. A step further, as an extension of iOO, the evolutionary OO (eOO) analyzes the workload patterns among consecutive phases and adjusts the scheduling based on the patterns. We develop a series of algorithms to partition and merge workload for efficient scheduling. We use a dynamic scenario of scheduling multitask scientific workloads to a group of virtual clusters on Amazon EC2 cloud. The experiment results show that the eOO approach achieves up to 30% performance speedup regarding task throughput, compared with Monte Carlo [23] and Blind Pick. As far as we know, this is the first time an OO is applied in an evolutionary way for dynamic optimization scenarios to meet special requirements of cloud workload scheduling.

The rest of the paper is organized as follows. Section II provides a brief description of the dynamic multitasking workload scheduling problem and existing methods. In Section III, we describe our proposed eOO technique that iteratively applies the OO for scheduling workloads on virtual clusters of a cloud. The experimental results, performance evaluation, and comparative analysis are presented in Section IV that also includes detailed information on the applications and system configurations. The related work is reviewed in Section V, and we provide concluding remarks in Section VI.

II. DYNAMIC WORKLOAD SCHEDULING

In this section, first, we introduce our dynamic multitasking-scheduling model. Thereafter, the necessity of using simulations to is presented. To follow, three existing approaches, Monte Carlo, Blind Pick, and iterative Ordinal Optimization, are introduced. To each readership, Table 1 summarizes the most frequently used symbols, notations, and definitions.

<table>
<thead>
<tr>
<th>Notations</th>
<th>Definition and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_i$</td>
<td>Throughput at the $i$-th scheduling period</td>
</tr>
<tr>
<td>$S_i$</td>
<td>The $i$-th scheduling period</td>
</tr>
<tr>
<td>$\Delta \bar{\delta}(t)$</td>
<td>New generated workload for VC, at time $t$</td>
</tr>
<tr>
<td>$\bar{\delta}(t)$</td>
<td>Remaining workload for VC, at time $t$</td>
</tr>
<tr>
<td>$\bar{\delta}(t)$</td>
<td>Total workload for VC, at time $t$</td>
</tr>
<tr>
<td>$CET(t_{i1}, t_{i2})$</td>
<td>Makespan in VC, from time $t_{i1}$ to $t_{i2}$</td>
</tr>
<tr>
<td>$EET(t_{i1}, t_{i2})$</td>
<td>Makespan from time $t_{i1}$ to $t_{i2}$</td>
</tr>
<tr>
<td>$c \in C$</td>
<td>Index of a VC or the number of VCs</td>
</tr>
<tr>
<td>$i \in I$</td>
<td>Index of time or the last scheduling point</td>
</tr>
<tr>
<td>$\theta(k)$</td>
<td>Number of VMs in VC, that are allocated at time $t_i$ for the $S_i$ stage</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Number of VMs available for scheduling</td>
</tr>
<tr>
<td>$\beta(k)$</td>
<td>Expected execution time in VC, at time $t_i$</td>
</tr>
<tr>
<td>$\gamma(k)$</td>
<td>Job processing rate in VC, at time $t_i$</td>
</tr>
<tr>
<td>$r(k)$</td>
<td>Remaining execution time in VC, at time $t_i$</td>
</tr>
</tbody>
</table>

A. DYNAMIC MULTITASKING WORKLOAD SCHEDULING MODEL

In this model, we define the task class as a set of tasks that are of the same type and can be executed concurrently. Suppose there are $C$ task classes in all, and the index is denoted as $c$, $c \in [1, C]$. Tasks within one task class can be either interdependent, such as scientific workflow, or independent of each other, such as scientific simulations with multiple parameters. Tasks across different task classes are independent. Virtual Machines (VMs) are the scheduling and computing units built on top of the physical machines. Given a fixed number of VMs, our scheduling method organizes them into $C$ groups, each group serves for one task class. Each group
The real workload at \( t \) is the sum of the remaining workload of the previous stages and the newly generated workload at time \( t \). We use \( \delta_c(t) \) and \( \delta_i(t) \) to denote the real and remaining workload at time \( t \) for the \( c \)-th VC, which gives us:

\[
\delta_c(t) = \delta_0(t) + \Delta\delta_c(t) \quad c \in [1, C].
\]

The aforementioned results are \( \delta(t) = [\delta_1(t), \delta_2(t), \ldots, \delta_i(t), \ldots, \delta_C(t)]^T \).

The tasks arriving at different VCs are time-variant. Our target is to look for a time series of schedules:

\[
\theta(t) = [\theta(t_1), \theta(t_2), \ldots, \theta(t_i), \ldots, \theta(t_f)],
\]

to maximize the throughput of the whole workload. The throughput is defined as the ratio of the total number of tasks finished and the time spent by the VCs to process the tasks. To calculate throughput, we need to find out the schedule time points, as well as the corresponding schedules for each time point.

From time point \( t_i \) to \( t_{i+1} \), the phrasal optimal throughput given the workload and phase can be represented by the following:

\[
T_{i*} = \max T(\theta(t_i)|\delta(t), S_{i-1})
= \max T(\theta(t_i)|\delta(t), t_i, t_{i-1}) \forall [t_i, t_{i-1}), \theta(t_i) \in U
\]

We first introduce the concepts of Class Execution Time (CET) and Effective Execution Time (EET). Suppose that \( t_{i,0} \) and \( t_{i,1} \) are the time points of two consecutive workload stages, as shown in Fig. 2. \( CET_c(t_{i,0}, t_{i,1}) \) is the runtime from the starting point \( t_{i,0} \) to the finishing time when all the tasks in task class \( c \) are finished. We show \( CET_c(t_{i,0}, t_{i,1}), c = \{1, 2, C\} \) in Fig. 2 by the three double end-arrow dashed lines. If \( CET_c(t_{i,0}, t_{i,1}) \) is larger than the scheduling period length \( t_{i,1} - t_{i,0} \) which means all the tasks in this task class cannot be finished within \( [t_{i,0}, t_{i,1}] \), then \( t_{i,1} - t_{i,0} \) is used as CET and the rest tasks are rolled over to \( \Delta\delta_c(t_{i,1}) \). EET\((t_{i,0}, t_{i,1})\) on the other hand, means the longest execution time across all

![FIGURE 2. Demonstration of the stacked workload of different VC, given in a timeline. Each rectangular box represents the workload generated for the corresponding virtual cluster. Between schedule periods \([t_1, t_2]\), there are time points at which new workloads are generated (or arrive). The time points are represented as \( t_1, t_2, \ldots, t_N \).](image)
of the task classes within \([t_i, 0, t_i, 1]\). It can be formally defined as follows.

\[
\text{EEF}(t_{i,n}, t_{i,n+1}) = \begin{cases} \\
\max \text{CET}_c(t_{i,n}, t_{i,n+1}) & \text{if } \text{CET}_c(t_{i,n}, t_{i,n+1}) < t_{i,n+1} - t_{i,n} \\
 t_{i,n+1} - t_{i,n} & \text{if } \text{CET}_c(t_{i,n}, t_{i,n+1}) \geq t_{i,n+1} - t_{i,n} \\
\end{cases}
\]

As shown in Fig. 2, \(EET\) \((t_{i,1}, t_{i,0})\) equals to \(CET_1\) \((t_{i,1}, t_{i,0})\). Therefore, the period throughput from \(t_i\) to \(t_{i+1}\) is calculated by:

\[
T_i^* = \frac{\sum_{j=1}^{N_i-1} \sum_{c=1}^{C} (\delta_c(t_{j-1}) - \delta_c(t_j))}{\sum_{j=1}^{N_i-1} EET(t_{ij}, t_{i,j+1})}
\]

Let \(\delta_c(t_i)\) be the number of tasks in the \(c\)-thVC at time \(t_i\), \(p_c(t_i)\) be the expected execution time for each VM in the \(c\)-th VC at time \(t_i\). Consequently, \(\beta_c(t_i) = \theta_c(t_i)/p_c(t_i)\) represents the corresponding job processing rate, where \(\theta_c(t_i)\) is the number of VMs in VC \(c\) that are allocated at time \(t_i\) and \(r_c(t_i) = \delta_c(t_i)/\beta_c(t_i)\) is the remaining execution time in the \(c\)-thVC at time \(t_i\).

The task throughput is used as the performance metric for the schedules. Given a schedule \(\theta(t) = \{\theta_1(t), \theta_2(t), ..., \theta_j(t), ..., \theta_j(t)\}\), the task throughput is calculated as the total number of finished tasks divided by the total task execution time (or the makespan). At different time periods, different schedules may be applied. All of the candidate schedules at each successive time period form a schedule space \(U\). The cardinality of \(U\) is calculated by the following expression:

\[
u = (\theta - 1)!/[((\theta - C)!)(C - 1)!],
\]

where \(\theta\) is the total number of VMs used in \(C\) virtual clusters. The parameter \(u\) counts the number of ways to partition a set of \(\theta\) VMs into \(C\) nonempty clusters.

For example, if we use \(\theta = 20\) VMs in \(C = 7\) VCs for seven task classes, then we need to assess \(u = 27,132\) possible schedules to search for the best schedule at each time \(t_i\). Each schedule \(\theta(t_i)\) takes the form of seven dimensional tuple, having a sum equal to 20, such as \([2, 3, 3, 5, 2, 3, 2]\) or \([6, 2, 1, 3, 2, 4, 2]\). The search space \(U\) is very large given the large number of VMs \(\theta\), which leads to the ineffectiveness of the scheduling method. Therefore, the schedule search space must be reduced significantly, especially in a dynamic environment.

### B. SIMULATION-BASED OPTIMIZATION METHODS

In this section, we introduce a series of existing simulation-based optimization solutions for this problem, namely Monte Carlo Simulation, Blind Pick, and iOO.

Simulations are needed due to the fact that the expected execution time \(p_c(t_i)\) is unknown in real-time. The value is stochastic and may be subjected to unknown runtime conditions, such as the contention of other virtual resources.

<table>
<thead>
<tr>
<th>Algorithm 1 Monte Carlo Simulation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
</tr>
<tr>
<td>(U = {\theta - {\theta_0, \theta_1, ..., \theta_j, ..., \theta_j}})</td>
</tr>
<tr>
<td>(\delta = {\delta(t_0), \delta(t_1), ..., \delta(t_i), ..., \delta(t_{\infty})})</td>
</tr>
<tr>
<td>(\alpha)</td>
</tr>
<tr>
<td><strong>Output:</strong></td>
</tr>
<tr>
<td>The best (\theta^*(t)) for use at each schedule time (t)</td>
</tr>
<tr>
<td><strong>Procedure:</strong></td>
</tr>
<tr>
<td>1. Random sample (U) as per the ratio (\alpha); Generate search space</td>
</tr>
<tr>
<td>2. Estimate the overhead time of using Monte Carlo method: (O_{MC})</td>
</tr>
<tr>
<td>3. Initialize schedule period valuable (i = 1);</td>
</tr>
<tr>
<td>4. while ((i * O_{MC}) &lt; t_i) do</td>
</tr>
<tr>
<td>5. (t_i = (i - 1) * O_{MC}, t_{i+1} = i * O_{MC})</td>
</tr>
<tr>
<td>6. Load workload (\delta(t)) between ([t_i, t_{i+1}]);</td>
</tr>
<tr>
<td>7. for each (\theta) in (U) do</td>
</tr>
<tr>
<td>8. for each (j) in ([0, N]) do</td>
</tr>
<tr>
<td>9. for each (VC) in ([1, C]) do</td>
</tr>
<tr>
<td>10. Simulate (p_c(t)) (N) times to get an average (P_c(t_i));</td>
</tr>
<tr>
<td>11. (\delta(t) = \delta(t) + \Delta\delta(t));</td>
</tr>
<tr>
<td>12. if ((\delta(t_{i+1}) - (t_{ij}, t_{ij+1}) &gt; P_c(t_i))) Not all the tasks finished</td>
</tr>
<tr>
<td>13. (\delta(t_{i+1}) = \delta(t_i) - P_c(t_i));</td>
</tr>
<tr>
<td>14. (CET(t_{i+1}, t_{ij+1}) = t_{ij+1} - t_i);</td>
</tr>
<tr>
<td>15. else</td>
</tr>
<tr>
<td>16. (\delta(t_{i+1}) = \delta(t_i));</td>
</tr>
<tr>
<td>17. end if;</td>
</tr>
<tr>
<td>18. (CET(t_{i+1}, t_{ij+1}) = (\delta(t_i) - (t_{ij}, t_{ij+1}) &gt; P_c(t_i)));</td>
</tr>
<tr>
<td>19. end for;</td>
</tr>
<tr>
<td>20. (\text{Makespan}(\theta(t)) = EET(t_{i+1}, t_{i}) + ... + EET(t_{i+1}, t_{i+1});)</td>
</tr>
<tr>
<td>21. end for;</td>
</tr>
<tr>
<td>22. if ((\text{Makespan}(\theta(t)) &lt; \text{Makespan}(\theta^*(t)))) A better schedule</td>
</tr>
<tr>
<td>(\theta^*(t) = \theta(t));</td>
</tr>
<tr>
<td>23. (T^<em>(t) = (\delta(t_0) - \delta(t_{i+1}) + \delta(t_{i+1})); (\delta(t_{i+1}) + ... + \delta(t_{\infty})))/(\theta^</em>(t));</td>
</tr>
<tr>
<td>24. end if;</td>
</tr>
<tr>
<td>25. end for;</td>
</tr>
<tr>
<td>26. (i = i + 1);</td>
</tr>
<tr>
<td>27. end while</td>
</tr>
</tbody>
</table>

Moreover, the random runtime workload distribution in the \(c\)-thVC also has a huge impact on its value.

For a given number of cloud VM instances, configuration (2ECU in Amazon EC2, 1.7GB Memory), VM utilization value, and failure rate, we evaluate the execution time distribution of a particular task class with a number of tasks, and correspondingly create a table.

For example, the table tells that the execution time is a normal distribution \((X \sim N(20, 5^2))\) when running \([100, 110]\) tasks of task class two with 10 VMs, each being small Amazon EC2 instance, failure rate being 0 and utilization being in a range of \([70\%, 80\%]\). Then, before each experiment runs, if we have profiled the average VM utilization is 75%, and there are 105 tasks and 10 VMs, then we must sample the normal distribution above to estimate \(p_c(t_i)\), and apply this sample value to estimate the throughput of each schedule, and finally choose the best schedule for use. Algorithm 1 below introduces the process.

To yield such an estimation, however, multiple simulation runs must be conducted and an observed average is to be used for \(p_c(t_i)\). This is a typical Monte Carlo simulation method that is known to be time consuming.
1) Monte Carlo Method: Suppose that the overhead of using the Monte Carlo method is denoted by $O_M$, which is the length of the scheduling period. Thereafter, we search through all of the time points between $t_i$ and $t_i+1$, when new workloads are generated, to calculate the CET for each VC as shown in Line 14 and 18. This step is followed by calculating the EET during the scheduling period, as shown in Line 20. Following this, the Makespan and throughput are calculated.

2) Blind Pick Method: Instead of searching through the whole schedule space $U$ as done in the Monte Carlo method, the Blind Pick (BP) method, randomly selects a portion of the schedules only within $U$ for evaluation. The ratio to be selected is defined by a value $\alpha$ ($0 \leq \alpha \leq 1$). The algorithm of applying BP is exactly the same as Monte Carlo, except for the scheduling sample space $U$ and scheduling overhead. The length of the scheduling period is $\alpha \times O_M$.

3) Iterative Ordinal Optimization Method: The OO, a sub-optimal low overhead scheduling method is thoroughly introduced in [15]. Different from the aforementioned methods, the OO uses a rough model ($n$ repeated runs) to generate a rough order of all of the schedules in $U$, and uses the accurate model ($N$ repeated runs, $n \ll N$) to evaluate the top-schedules in the rough order list. For the values of $s$, the readers are referred to Section III.B.

The OO method contains two steps. The rough evaluation step evaluates $u$ schedules, each one being repeated $n$ times, and the overhead becomes $u \times O \times n$. Suppose $O$ denotes the time needed to evaluate one schedule, then the second stage will take $s \times O \times N$ amounts of time.

We have proven in our previous paper [40] that the OO simulation time is shorter than other two methods that enable us to develop an iterative OO (iOO) method. In other words, the iOO applies the OO method iteratively in a multi-stage scheduling manner, with each stage being a new OO routine.

The iOO approach exhibited superior performance in the research paper, especially in the highly fluctuated workload cases due to the reduced overhead, resulting in shorter scheduling periods and fine-grained scheduling. The length of the iOO scheduling period can be determined as $u \times O \times n + s \times O \times N$.

However, the iOO does not consider the characteristics of workloads, namely the similarity of the workloads in consequent stages of a multi-stage scheduling problem. In the next section, we introduce the evolutionary Ordinal Optimization (eOO) that extends the iOO method along those lines.

III. EVOLUTIONARY ORDINAL OPTIMIZATION

An intuitive interpretation of the eOO is the assumption that the simulation runs must be allocated based on the characteristics of the workloads. For example, fluctuating workload requires more intermediate scheduling time points. Because each of the short scheduling phases satisfies small periods of the workload, the eOO delivers better overall performance.

However, one disadvantage of the eOO method is that the short simulation phase leads to less simulation runs of the $p_s(t_i)$, which on the contrary, might decrease the performance of each of the single schedule period. In this section, we demonstrate the methodologies and effectiveness of the eOO method, and propose a solution to partition and merge the scheduling periods based on the fluctuations within the workload.

A. EVOLUTIONARY ORDINAL OPTIMIZATION

Before we step into any further detail, we define the concept of workload pattern. At time $t_0$ in Fig. 3, one batch of workload for the seven VCs arrives. This workload pattern is very similar to the workload patterns at $t_1$ and $t_2$. To give a formal definition, suppose that there are two batches of workloads at time $t'$, $t'' \in [t_i, t_{i+1}]$, i.e., $\delta(t') = (\delta_1(t'), \ldots, \delta_i(t'), \ldots, \delta_C(t'))$ and $\delta(t'') = (\delta_1(t''), \ldots, \delta_i(t''), \ldots, \delta_C(t''))$, respectively, where $\delta_i(t')$ denotes the tasks for VC $c$ at time $t'$. The similarity pattern between $\delta(t')$ and $\delta(t'')$ can be defined as:

$$\text{Sim}(\delta(t'), \delta(t'')) = \frac{\sum_{c=1}^{C} (\delta_i(t') \times \delta_i(t''))}{\sqrt{\sum_{c=1}^{C} \delta_i(t')^2 \times \sum_{c=1}^{C} \delta_i(t'')^2}}$$

Intuitively, a high similarity pattern leads to the merger of two scheduling periods, while a low similarity pattern leads to a scheduling period partition. In Fig. 3, we would merge $[t_0, t_1]$, $[t_1, t_2]$, and $[t_2, t_3]$ as a single scheduling period, when supposing that $t$ is the scheduling period of any method proposed in the previous section. However, such a merge does not apply to the period $[t_3, t_3 + t]$, where the two batches of workloads are dramatically different from each other. Therefore, we must divide the scheduling period adaptively into smaller periods for separate scheduling.

![FIGURE 3. Demonstration of various workload characteristics. The dashed-line rectangular box with seven solid lines are the seven workloads for the C = 7 VCs. There are five batches of workloads arrive at time $t_0$, $t_1$, $t_2$, $t_3$ and $t_4$ for each of the virtual clusters.](image)
Algorithm 2 Scheduling Period Partition

Input: All workloads in \([t_0, t_1]\) denoted by \(\mathcal{A}(t_0) = \{\mathcal{A}(t_0), \mathcal{A}(t_0+1), \ldots, \mathcal{A}(t_n)\}\).

Output: Decision point (DP) set in \([t_0, t_1]\).

Procedure:
1. \(DP \leftarrow \text{null};\)
2. if \((t_i - t_0 < w)\) //Interval is too small, cannot be partitioned anymore
3. Add \(t_i\) into \(DP\);
4. return;
5. end if;
6. for each workload \(\mathcal{A}(t_i)\) in \(\mathcal{A}(t_0, t_1)\)
7. for each workload \(\mathcal{A}(t_i)\) in \(\mathcal{A}(t_0, t_n)\) & & \(k > l\) do
8. if \((\max(\text{Sim}(\mathcal{A}(t_i), \mathcal{A}(t_l))) < \alpha)\) //small similarity
9. Partition \((t_0, t_i + (t_i - t_0) / 2)\); //partition left half
10. Partition \((t_i + (t_i - t_0) / 2, t_1)\); //partition right half
11. end if;
12. end for;
13. end for;
14. Sort and return \(DP\).

and \([t_i+1, t_{i+2}]\), respectively, where \(i' \in [t_i, t_{i+1}]\) and \(i'' \in [t_i+1, t_{i+2}]\). We make similarity analysis among all of the possible pairs as mentioned above. If the least similarity value is sufficiently large, then the two scheduling periods must be merged. Such a merge operation leads to a sufficiently large simulation period of \([t_i, t_{i+2}]\) that promises a longer simulation period for the subsequent period.

In Algorithm 2 and Algorithm 3, we show the details of the scheduling period merge and partition based on the workload similarities. In general, for Algorithm 2, if the maximum similarity value between any two pairs of workloads in \([t_i, t_{i+1}]\) is less than \(\alpha\), then the interval will be partitioned into two parts. For the aforementioned case, we use a recursive algorithm, as shown in Line 9 and Line 10, to further analyze the two sub-intervals. The analysis stops when either the sub-interval is smaller than a threshold value \(w\), or all of the workload patterns are similar in the interval that is being checked. The algorithm outputs all of the decision time points \(DP\).

B. PARAMETER CONFIGURATION

Unknown parameters must be finalized before the experimental runs. These are the simulation runs \(N\) and \(n\), size of selected set \(s_i\) in iOO, size of the selected set \(s_{e}\) using eOO, size of the selected set \(s_{BP}\) using BP, \(\alpha\) and \(\beta\) in Algorithm 3 and Algorithm 4.

The BF Monte Carlo repeats itself \(N\) times to estimate the average value of each \(p_i(t_i)\). As a common practice to achieve reliable simulation accuracy, Monte Carlo usually applies a large simulation runs, say \(N = 1000\), for each single individual schedule for large-scale scientific workflow scheduling problems [42].

The rough model \(n\) of the iOO is also determined by the statistic theory. Based on the central tendency theory, it takes two orders of magnitudes more runs to improve one order of magnitude of the estimation accuracy. When applying \(n = 10\) runs in the rough model, which is two orders of magnitude less simulation runs than \(N\), would lead to one order of higher estimation error. For example, suppose the estimation error of BF Monte Carlo stays in a range from 0.1 to 0.9, the estimation error of iOO would be in a range from 1 to 9. Details analysis of and a formal proof of the relationship between the simulation time and simulation accuracy can also be referred to the work [42].

Selection set size of each method, BP, iOO and eOO is different. The BP method picks a fixed ratio \(\alpha\) that is normally larger than 80%. As we mentioned earlier, the iOO and eOO are two-stage scheduling algorithms. Their selection sets are determined by the time left for the second scheduling period. For example, if each scheduling period of iOO equals to \(t = 100\) seconds and the rough evaluation of all schedules takes 10 seconds. Suppose a precise evaluation of one schedule takes 3 seconds, the maximum selection set size for iOO would be 30.

Around ten thousand groups of simulated data were generated randomly based on the distribution of the workloads. We plot the histogram of the statistic analysis of these workloads as shown in Fig. 4. Six consecutive ranges of pair-wise similarity degree values are used as the x-axis and the total number of workload pairs that shows the similarity degree is plotted as the y-axis. This method classifies the similarities among all the workloads in a few categories, and identifies quantitatively the similar value threshed we use to determine
whether one pair of workload being similar or not. We set the upper third quantile of the distribution as $\alpha$ and the lower third quantile as $\beta$. The results are reported in Fig. 4.

![FIGURE 4. The cumulative number of workloads and the similarities between all of the workload pairs over the similarity degree values.](image)

C. THE eOO FOR DYNAMIC WORKLOAD SCHEDULING

If the scheduling overhead is too high and the scheduling itself takes longer than the period of workload and resource changing, then the dynamic workload in the multitasking scheduling problem is difficult to capture. However, applying the OO in an evolutionary manner, leads to the full investigation of the workload patterns. For the highly dynamic and fluctuating workloads, the eOO adaptively partitions into small pieces, to generate the fine-grained schedules. Although the short-period scheduling means less accuracy, the gain in the adaptivity to such kinds of workloads, outweighs the minor inaccuracies in the solutions.

On the contrary, the eOO produces course-grained schedules for stable workloads. In this way, the merged stages lead to more simulation time for a subsequent stage that improves the accuracy. We illustrate the method graphically in Fig. 5.

![FIGURE 5. Illustration of OO adaption to dynamic workloads.](image)

D. OPTIMAL COMPUTING BUDGET ALLOCATION BASED SCHEDULING

All of the scheduling methods, namely Monte Carlo, Blind Pick, iOO, eOO can be further improved by combining them with the Optimal Computing Budget Allocation (OCBA) [7].

The OCBA is a simulation-based optimization approach that allocates computing resources effectively among all of the candidate schedules. Simply speaking, rather than equally allocate simulation time among all of the schedules, the OCBA allocates more simulation time to the schedules that show better overall performances in their early simulation stages. Such an exercise removes the potentially poor schedules at an early stage, and gives more observations to the promising schedules. In our experimental evaluations, we apply the OCBA method on each of the schedule stage derived by the eOO, and compare the performances of all of the methods.

IV. EVALUATIONS, RESULTS, AND DISCUSSIONS

In this section, detailed experimental design and results are presented for the virtual cluster allocation in multitasking workload scheduling applications.

A. EXPERIMENTAL SETTINGS

The cloud experiments are carried out using 16, 32, 64, and up to 128 standard VM instances on the Amazon EC2 platform. Each instance is by default, a small one with 1.7 GB of memory, one EC2 Compute Unit (one virtual core with one EC2 Compute Unit), 160 GB of local instance storage, a 32-bit platform on deployed Linux Base with EBS boot, and a 32-bit architecture with Amazon EC2 Tools shown in Fig. 6.

The benchmarking applications used are similar to [8]. The authors report a benchmark application suite includes bit-reversals, merge-sort, matrix multiplication, a trace-driven simulation, partitioning meshes, k-nearest neighbor classification, and AdaBoost algorithm. These seven applications correspond to our seven tasks classes, which need seven virtual clusters to host and execute.
Up to 10,000 tasks were concurrently executed. Each task corresponding to one benchmark application has a variable amount of input dataset size. The total amount of data used in the 128-VM scenarios approximately exceeds 1.6 TB.

We define four typical workload types by using, say, LFLD to represent Low Frequency Low Difference (LFLD) workload. This means the workload comes slowly while most of the workload patterns are alike. Similarly, the three other workload types are LFHD, HFLD and HFHD. In the HFHD case, the average arrival rate of the workload comes up to 500 tasks/second. The high velocity and volume of data used justifies a testing-scale big data analytic in these experiments.

Even though the data amount tested here is not extremely large, we argue that the method can still be extrapolated to a larger scale of hundred TB or even PB of data due to the loosely-coupled multitasking property.

The hierarchical structure of our experiments is shown in Fig. 7. Each server notation is an Amazon EC2 instance. There are 7 VCs denoted by the dashed-rectangular boxes. There is one level-2 seed server in each of the VC to ensure at least one VM is running to process the requests from level-1 seed server. It is also used to balance the workloads into the internal VMs. This is a dedicated VM that resides in each of the VC.

**B. APPLICATION SCENARIOS**

Two major application scenarios are depicted in Fig. 8 below. We describe both of them in detail.

**Scenario 1: Simulation before execution.** In the upper part of Fig. 8, one simulation stage is conducted before each experimental run. Before execution stage 1, there is sufficient time for a simulation to find a good allocation schedule from $t_0$ to $t_1$, and the schedule is applied at $t_1$. No simulation is carried out during execution stage 1 until $t_2$. This scenario is commonly seen in the job scheduling of traditional parallel and distributed computing systems and spot instances in Amazon EC2. For example, in using spot instances, there is sufficient time for simulation before the price reaches an acceptable level, and execution stage can follow the simulation.

**Scenario 2: Simulation while execution.** In the lower part of Fig. 8, simulation and execution stages overlap. At time $t_0$, a random schedule is used for the time being, while the workload information between $[t_1, t_2]$ is used to generate a better quality schedule to be used at time $t_1$. Similar scheduling is conducted repeatedly until the end of experiment. Most of the real-time multitasking scheduling applications are carried out based on Scenario 2.

**C. COMPARATIVE STUDIES ON SCENARIO 1**

We compare the case studies of Scenario 1, on the above-mentioned four types of workloads. The results are reported in Fig. 9 (a)-(d).

It can be observed that the workload variation has a very limited impact on the three methods. The BF Monte Carlo is always the best while the Blind Pick is always the worst among all. The OO method lies in between. In Table 2, we

<table>
<thead>
<tr>
<th>Cluster Size</th>
<th>Methods</th>
<th>LFLD</th>
<th>LFHD</th>
<th>HFHD</th>
<th>HFHD</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Monte Carlo</td>
<td>6.19%</td>
<td>5.50%</td>
<td>1.19%</td>
<td>0.80%</td>
</tr>
<tr>
<td>32</td>
<td>Monte Carlo</td>
<td>3.18%</td>
<td>3.32%</td>
<td>2.53%</td>
<td>6.48%</td>
</tr>
<tr>
<td>64</td>
<td>Monte Carlo</td>
<td>1.29%</td>
<td>3.51%</td>
<td>4.01%</td>
<td>2.96%</td>
</tr>
<tr>
<td>128</td>
<td>Monte Carlo</td>
<td>1.07%</td>
<td>1.50%</td>
<td>0.04%</td>
<td>0.13%</td>
</tr>
<tr>
<td>16</td>
<td>Blind Pick</td>
<td>-1.03%</td>
<td>-3.67%</td>
<td>-2.97%</td>
<td>-4.73%</td>
</tr>
<tr>
<td>32</td>
<td>Blind Pick</td>
<td>-2.27%</td>
<td>-0.83%</td>
<td>-3.78%</td>
<td>-5.04%</td>
</tr>
<tr>
<td>64</td>
<td>Blind Pick</td>
<td>-2.58%</td>
<td>-4.96%</td>
<td>-4.78%</td>
<td>-2.31%</td>
</tr>
<tr>
<td>128</td>
<td>Blind Pick</td>
<td>-1.46%</td>
<td>-2.07%</td>
<td>-0.68%</td>
<td>-2.27%</td>
</tr>
</tbody>
</table>
FIGURE 9. Comparative studies of application scenario 1. (a) LFLD. (b) LFHD. (c) HFLD. (d) HFHD.

The aforementioned behavior is straightforward. Given sufficient simulation time, the BF Monte Carlo simulates longer than any other methods to evaluate \( p_c(t_i) \), which undoubtedly achieves the best performance because of its precise evaluation. Random selection of the Blind Pick method leads to the worst performance. This Method takes the least of the simulation time; however, the performance is mediocre at best. The OO method cannot take advantage of the long simulation time allowed and has improvement over Blind Pick, but not as good as BF Monte Carlo.

D. COMPARATIVE STUDIES ON SCENARIO 2

The simulation time budget in Scenario 2 is limited. Consequently, continuous simulation and scheduling are performed simultaneously.

In this scenario, the BF Monte Carlo method with extended simulation is no longer the best, with lower throughput compared with the iOO and eOO methods. From Fig. 10(a)-(d), we compare the throughput of the methods using a variable size of cluster. The corresponding performance comparison using eOO compared with other methods are summarized in Table 3.

TABLE 3. Performance comparison of scenario 2.

<table>
<thead>
<tr>
<th>Cluster Size</th>
<th>Methods</th>
<th>LFLD</th>
<th>LFHD</th>
<th>HFLD</th>
<th>HFHD</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Monte Carlo</td>
<td>-5.88%</td>
<td>-12.12%</td>
<td>-17.92%</td>
<td>-7.12%</td>
</tr>
<tr>
<td>32</td>
<td>Monte Carlo</td>
<td>-8.51%</td>
<td>-10.43%</td>
<td>-17.71%</td>
<td>-10.31%</td>
</tr>
<tr>
<td>64</td>
<td>Monte Carlo</td>
<td>-9.56%</td>
<td>-12.31%</td>
<td>-22.26%</td>
<td>-9.63%</td>
</tr>
<tr>
<td>128</td>
<td>Monte Carlo</td>
<td>-16.00%</td>
<td>-13.05%</td>
<td>-18.68%</td>
<td>-29.06%</td>
</tr>
<tr>
<td>16</td>
<td>Blind Pick</td>
<td>0.98%</td>
<td>-18.18%</td>
<td>-14.41%</td>
<td>-7.98%</td>
</tr>
<tr>
<td>32</td>
<td>Blind Pick</td>
<td>-3.40%</td>
<td>-10.43%</td>
<td>-12.31%</td>
<td>-20.13%</td>
</tr>
<tr>
<td>64</td>
<td>Blind Pick</td>
<td>-5.98%</td>
<td>-19.13%</td>
<td>-15.09%</td>
<td>-16.50%</td>
</tr>
<tr>
<td>128</td>
<td>Blind Pick</td>
<td>-13.85%</td>
<td>-19.48%</td>
<td>-18.08%</td>
<td>-32.62%</td>
</tr>
<tr>
<td>16</td>
<td>iOO</td>
<td>-2.94%</td>
<td>-8.33%</td>
<td>-7.29%</td>
<td>-5.56%</td>
</tr>
<tr>
<td>32</td>
<td>iOO</td>
<td>-2.13%</td>
<td>-3.24%</td>
<td>-15.66%</td>
<td>-3.18%</td>
</tr>
<tr>
<td>64</td>
<td>iOO</td>
<td>-2.79%</td>
<td>-5.82%</td>
<td>-9.48%</td>
<td>-7.06%</td>
</tr>
<tr>
<td>128</td>
<td>iOO</td>
<td>-2.24%</td>
<td>-11.35%</td>
<td>-6.06%</td>
<td>-14.05%</td>
</tr>
</tbody>
</table>

The performance of all the four methods is similar when a small number of VMs is used. For example, the four methods perform similarly in the case of 16 VMs. However, the difference grows substantially as the number of VMs increase. In the case of 128 VMs, the eOO performs 32.62% and 29.06% better than the BP and the BF.

The relative performance of the BF Monte Carlo and BP varies given different types of workloads. In the LFHD and HFHD cases, the BP performs better; while in the LFLD and HFLD ones, the BF Monte Carlo slightly outperforms the BP. This is caused by the low overhead of BP. The BF Monte Carlo with long simulation time and high overhead, is inapplicable in Scenario 2.

The advantage of the eOO over the iOO resides in its adaptability in mutating the scheduling periods. In Fig. 10(b) and Fig. 10(d), where the workloads exhibit high difference...
over time, the eOO performs much better with its capability to partition one period into two or merge two neighboring periods into one. This is a workload pattern-aware method; therefore, it performs much better in highly dynamic workload cases.

E. SIMULATED COMPARATIVE STUDIES WITH OCBA

As mentioned previously (Section III.D), the OCBA attempts to judicially allocate the computing budget among all the multiple schedules, which breaks the equal budget allocation assumption implemented in the iOO and eOO methods. In simple words, the more a schedule shows better performance in a set of simulation runs, the more follow-up simulation runs the schedule tends to receive. By employing such a methodology, the potentially better schedules are evaluated more frequently and accurately, which translates into better system performance. Without explicitly mentioned, the OCBA used in this section is referred to using the eOO method by replacing its equal schedule simulation allocation time with the OCBA algorithm.

Unlike other methods, the OCBA calculates the simulation time for each schedule and adjusts it in runtime. In a real cloud-computing environment, this solution faces a difficulty by not being able to effectively predict the simulation cycles. Instead, we use simulations only to prove the effectiveness of this method. In our future works, we will explore by studying more scenarios to justify the applicability of this method in real cloud computing environments. The comparative results are shown in Fig. 11 by simulating a cluster of using 32 VMs. The performance metric is the makespan which aggregates all of the effective compute resource usage time, denoted by the \(EET\) (as detailed in Section II.A).

In Table 4, similar performance comparison of using OCBA compared with other methods is given. OCBA improves up to 29.69% performance than other methods. This improvement is observed mainly in HFHD workload. This validates our assumption again that, our methods are appealing to highly fluctuating workloads. We also demonstrate the effectiveness of using the OCBA method over all other methodologies.

<table>
<thead>
<tr>
<th>Method</th>
<th>LFLD</th>
<th>LFHD</th>
<th>HFLD</th>
<th>HFHD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monte Carlo</td>
<td>12.64%</td>
<td>18.33%</td>
<td>16.54%</td>
<td>29.68%</td>
</tr>
<tr>
<td>Blind Pick</td>
<td>14.28%</td>
<td>16.53%</td>
<td>15.79%</td>
<td>29.65%</td>
</tr>
<tr>
<td>iOO</td>
<td>8.52%</td>
<td>11.11%</td>
<td>10.65%</td>
<td>23.73%</td>
</tr>
<tr>
<td>eOO</td>
<td>2.27%</td>
<td>3.63%</td>
<td>2.01%</td>
<td>7.14%</td>
</tr>
</tbody>
</table>

Overall, there is a 2% to 7% speedup using the eOO with OCBA compared to the eOO method only. Compared to other methods, such as the Monte Carlo and the Blind Pick in the HFHD workload scenario, the speedup can be as much as 29%. In other less dynamic workload scenarios, the speedup value varied between 12.6% and 18.3%.

V. RELATED WORK

We have witnessed an escalating interest in resource allocation for multi-task scheduling [21], [32]. Many classical optimization methods, such as opportunistic load balance, minimum execution time, and minimum completion time, are
FIGURE 11. Comparative studies of application Scenario 2 with OCBA. (a) LFLD. (b) LFHD. (c) HFLD. (d) HFHD.

Benoit et al. [4] designed resource-aware allocation strategies for divisible loads. Li and Buyya [18] proposed a model-driven simulation grid scheduling strategies. Zomaya et al. [20], [33] have proposed a hybrid schedule and a cooperative game framework. There are also studies on making various tradeoffs under various constraints and objectives for the workflow scheduling. Wieczorek [37] analyzed five facets that may have a major impact on the selection of an appropriate scheduling strategy, and proposed taxonomies for the multi-objective workflow scheduling. Prodan et al. [29] proposed a novel dynamic constraint algorithm, which outperforms many existing methods, such as SOLOS and BDLS, to optimize bi-criteria problems.


Ozisikyilmaz et al. [26] suggested an efficient machine learning method for system space exploration. Similar to the work described in [19], our work is also carried out by using the search size based reduction. In [28], [33], and [34], petri-net and data-driven based methods are shown to compose services via the mediator, which motivates us to design the architectural cloud-scheduling platform for multitasking workloads.

A few control-based strategies are also proposed [9], [27], [45] to manipulate the VM count for delivering optimal performance. Other related work, such as [22] and [44], focus on finding optimal price strategy for renting spot instances. The similarity of these works lies to relying on their individual statistic models. Our work, however, is a simulation-based strategy, which is applied to multi-stage scheduling problems.

Based on the theory of the OO, we proposed the eOO that has advantage in handling large-scale search space to solve the multitask scheduling problem for dynamic workloads. Ever since the introduction of OO in [14], one can search for a small subset of solutions that are sufficiently good and computationally tractable. Along the OO line, many OO based heuristic methods have been proposed [17], [36]. It quickly narrows down the solution to a subset of “good enough” solutions with manageable overhead. Typical and original applications of the OO include automated manufacturing [31], communications [38], power systems [1], and distributed computing systems [43]. Different selection rules of OO are compared in [16] to discuss the relative performance. Conclusion were made that no selection rule is absolutely better than the others under all circumstances. To the best of our knowledge, this paper reports the first successful attempt to apply the OO in an iterative and evolutionary fashion for the dynamic optimization scenarios to meet special requirements of the cloud workload scheduling.
VI. CONCLUSIONS
A low-overhead method for dynamic multitasking workload scheduling in elastic virtual clusters is introduced in this work. The advantage of the proposed low-overhead scheduling scheme is summarized below.

1) Modeling of multitasking workload scheduling in wide-area elastic virtualized clusters. Cloud computing allocates virtual cluster resources on demand. This model serves the cloud computing environments and proposes simulation-based method for optimization.

2) The OO methodology is enhanced to deal with dynamically evolving workloads. iOO, with lower scheduling overhead and eOO, with adaptivity in terms of variable scheduling interval to turbulent workloads, are introduced, respectively. These two enhancements to OO are agile enough to capture system dynamism and conduct real-time scheduling. The two methods improve the overall performance in fluctuating workload scenarios evidenced by the experimental studies.

3) Proposed methods are validated by real benchmark applications with large search space, VM resource constraints, and uncertainties. Very few previous works utilizes simulation-based method for dynamic multitasking workload scheduling. Our approach is simulation based which takes consideration of these factors and is more realistic to be applied in real-world cloud systems.

Ongoing work includes the establishment of new cloud infrastructure for enabling real-time large-scale data analysis using a variety of more benchmark applications with varying workloads. We also intend to release the simulation-based optimization approaches as profiling tools to enhance performance of scheduling multitasking workload on Amazon EC2 platform.

Acknowledgment
This work was supported in part by the Ministry of Science and Technology of China through the National 973 Basic Research Program under Grant 2011CB302505 and Grant 2011CB302506, in part by the National Natural Science and Technology of China through the National 973 Basic Research Program under Grant 2013CB302505, in part by the Ministry of Science and Technology of China through the National 973 Basic Research Program under Grant 2013CB302506 and Grant 2013CB302505, in part by the National Natural Science Foundation of China under Grant 61472200 and Grant 61233016, and in part by the National Science Foundation under Grant CCF-1016966.

REFERENCES
ZHANG et al.: Evolutionary Scheduling of Dynamic Multitasking Workloads

FAN ZHANG (S’08–M’12–SM’13) is currently a Post-Doctoral Associate with the Kavli Institute for Astrophysics and Space Research, Massachusetts Institute of Technology, Cambridge, MA, USA. He has been a Visiting Associate Professor with the Shenzhen Institute of Advanced Technology, Chinese Academy of Science, Shenzhen, China, since 2014. He received the Ph.D. degree from the Department of Control Science and Engineering, Tsinghua University, Beijing, China, in 2012, the B.S. degree in computer science from the Hubei University of Technology, Wuhan, China, and the M.S. degree in control science and engineering from the Huazhong University of Science and Technology, Wuhan. From 2011 to 2013, he was a Research Scientist with the Cloud Computing Laboratory, Carnegie Mellon University, Pittsburgh, PA, USA. He was a recipient of an Honorarium Research Funding Award from the University of Chicago and the Argonne National Laboratory (2013), the Meritorious Service Award from the IEEE Transactions on Service Computing (2013), and two IBM Ph.D. Fellowship Awards (2010 and 2011). His research interests include big-data scientific computing applications, simulation-based optimization approaches, cloud computing, and novel programming models for streaming data applications on elastic cloud platforms.

WEI TAN (M’12–SM’13) received the B.S. and Ph.D. degrees from the Department of Automation, Tsinghua University, Beijing, China, in 2002 and 2008, respectively. He is currently a Research Staff Member with the IBM T. J. Watson Research Center, Yorktown Heights, NY, USA. From 2008 to 2010, he was a researcher with the Computation Institute, University of Chicago, Chicago, IL, USA, and the Argonne National Laboratory, Lemont, IL, USA. At that time, he was the technical lead of the caBIG workflow system. His research interests include NoSQL, big data, cloud computing, service-oriented architecture, business and scientific workflows, and Petri nets. He has authored over 50 journal and conference papers, and a monograph entitled Business and Scientific Workflows: A Web Service-Oriented Approach (272 pages, Wiley-IEEE Press). He is an Associate Editor of IEEE Transactions on Automation Science and Engineering. He served on the Program Committee of many conferences and co-chaired several workshops. He was a recipient of the Best Paper Award from the IEEE International Conference on Services Computing (2011), the Pacesetter Award from the Argonne National Laboratory (2010), and the caBIG Teamwork Award from the National Institute of Health (2008). He is a member of the Association for Computing Machinery.

SAMEE U. KHAN (S’02–M’07–SM’12) is currently an Associate Professor with North Dakota State University, Fargo, ND, USA. He received the Ph.D. degree from the University of Texas at Arlington, Arlington, TX, USA, in 2007. His research interests include optimization, robustness, and security of: cloud, grid, cluster, and big data computing, social networks, wired and wireless networks, power systems, smart grids, and optical networks. His work has appeared in over 225 publications with two receiving best paper awards. He is a fellow of the Institution of Engineering and Technology and the British Computer Society.
KEQIN LI (M’90–SM’96) is currently a SUNY Distinguished Professor of Computer Science and an Intellectual Ventures Endowed Visiting Chair Professor with Tsinghua University, Beijing, China. His research interests are mainly in design and analysis of algorithms, parallel and distributed computing, and computer networking. He has over 300 research publications, and has received several best paper awards for his research work. He is currently on the Editorial Boards of IEEE Transactions on Cloud Computing and the IEEE Transactions on Computers.

ALBERT Y. ZOMAYA (M’90–SM’97–F’04) is currently the Chair Professor of High Performance Computing and Networking with the School of Information Technologies, University of Sydney, Sydney, NSW, Australia, where he is also the Director of the Centre for Distributed and High Performance Computing, which was established in 2009. He has authored over 500 scientific papers and articles, and has authored, co-authored, or edited over 20 books. He is an Editor-in-Chief of the IEEE Transactions on Computers and Springer’s Scalable Computing, and serves as an Associate Editor of 22 leading journals, such as, the ACM Computing Surveys and Journal of Parallel and Distributed Computing.

Prof. Zomaya was a recipient of the IEEE Technical Committee on Parallel Processing Outstanding Service Award (2011), the IEEE Technical Committee on Scalable Computing Medal for Excellence in Scalable Computing (2011), and the IEEE Computer Society Technical Achievement Award (2014). He is a Chartered Engineer, and a fellow of the American Association for the Advancement of Science and the Institution of Engineering and Technology (U.K.). His research interests are in the areas of parallel and distributed computing and complex systems.