

An Availability-Aware Task Scheduling Strategy for Heterogeneous Systems

Xiao Qin, *Member, IEEE*, and Tao Xie, *Member, IEEE*

Abstract—High availability is a key requirement in the design and development of heterogeneous systems where processors operate at different speeds and are not continuously available for computation. Most existing scheduling algorithms designed for heterogeneous systems do not factor in availability requirements imposed by multiclass applications. To remedy this shortcoming, we investigate in this paper the scheduling problem for multiclass applications running in heterogeneous systems with availability constraints. In an effort to explore this issue, we model each node in a heterogeneous system using the node's computing capability and availability. Multiple classes of tasks are characterized by their execution times and availability requirements. To incorporate availability and heterogeneity into scheduling, we define new metrics to quantify system availability and heterogeneity for multiclass tasks. We then propose a scheduling algorithm to improve the availability of heterogeneous systems while maintaining good performance in the response time of tasks. Experimental results show that our algorithm achieves a good trade-off between availability and responsiveness.

Index Terms—Availability constraints, heterogeneous systems, multiclass applications, scheduling, resource allocation.

1 INTRODUCTION

OVER the last decade, heterogeneous systems have been widely used for scientific and commercial applications [9]. To improve the performance of applications running in heterogeneous systems, past research has developed a wide variety of scheduling algorithms for heterogeneous computing systems [8], [30]. Dogan and Özgüner developed reliable matching and scheduling algorithms for tasks with precedence constraints in heterogeneous distributed systems [8]. Srinivasan and Jha incorporated the reliability cost, defined to be the product of processor failure rate and task execution time, into scheduling algorithms for tasks with precedence constraints [29]. Ranaweera and Agrawal proposed a scalable scheduling scheme called Scalable Task-Duplication-based scheduling for Pipeline execution (STDP) for heterogeneous systems [20]. The objective of scheduling algorithms is to map tasks onto nodes and order their execution in a way to optimize overall performance.

In scheduling theory, the basic assumption is that all machines are always available for processing [23]. This assumption might be reasonable in some cases, but it is not valid in scenarios where there exist certain maintenance requirements, breakdowns, or other constraints, which make the machines unavailable for processing [23]. Examples of such constraints can be found in many application areas. For instance, computational nodes in heterogeneous

systems need to be periodically maintained to prevent malfunctions [14]. In this study, availability is defined as the ratio of the total time a computing node is functional during a given interval. The performance of a heterogeneous system will be degraded if one or multiple nodes are out of order due to random breakdown or preventive maintenance. On the other hand, however, nowadays, many high-performance applications require computing platforms with high availability [2], [21], [22], [26], [24]. Military applications, 24 × 7 healthcare applications, international business applications, and the like demand extremely high-availability services since severe damage or fatal errors could occur when even only one computing node becomes unavailable [2]. As such, a scheduling strategy for heterogeneous systems has to factor in availability to deal with maintenance activities and unexpected failures.

A multiclass application consists of tasks of multiple classes which are characterized by their distinctive arrival rates, execution time distributions, and availability requirements. The issue of scheduling multiple classes of tasks with availability constraints was raised by a wide range of real-world distributed applications such as scalable Web server systems [10], distributed heterogeneous servers [24], and general multiclass systems built on high-speed networks [11]. In many multiclass applications, incoming requests are immediately dispatched to one of a set of computing nodes, each of which independently executes a process running a local sequencing algorithm [10], [24]. Unfortunately, conventional scheduling algorithms [3], [4], [7], [10] for multiclass applications running in heterogeneous systems only concentrated on high throughput with the goal of reducing response times, completely ignoring the availability requirements of multiclass tasks. It is challenging, however, to achieve high throughput and high availability simultaneously because they are two conflicting objectives [2]. For example, it is unacceptable

• X. Qin is with the Department of Computer Science and Software Engineering, Dunstan Hall, Auburn University, Auburn, AL 36849-5347. E-mail: xqin@auburn.edu.

• T. Xie is with the Department of Computer Science, 5500 Campanile Drive, GMCS 544, San Diego State University, San Diego, CA 92182. E-mail: xie@cs.sdsu.edu.

Manuscript received 15 Mar. 2006; revised 7 Aug. 2006; accepted 26 Apr. 2007; published online 5 June 2007.

Recommended for acceptance by B. Veeravalli.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0098-0306.

Digital Object Identifier no. 10.1109/TC.2007.70738.

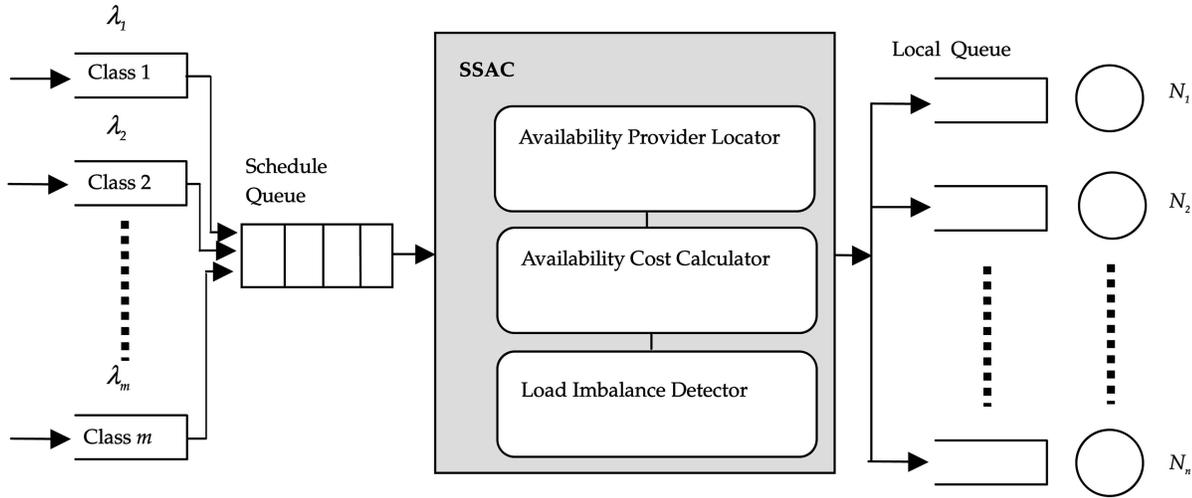


Fig. 1. System model of the SSAC strategy.

to assign a critical task with a high-availability requirement to a computing node that provides high speed and low availability. We argue that an ideal scheduling scheme has to guarantee the tasks' availability requirements while efficiently reducing the response times.

Some work has been done to investigate resource allocation schemes for tasks with availability constraints [22]. Smith introduced a mathematical model for resource availability and then proposed a method to maintain availability information as new reservations or assignments are made [26]. Adiri et al. addressed the scheduling issue in a single machine with availability constraints [1]. Qi et al. developed three heuristic algorithms to tackle the problem of scheduling jobs while maintaining machines [18] simultaneously. Very recently, Kacem et al. investigated a branch-and-bound method to solve the single-machine scheduling problem with availability constraints [13]. Lee studied the two-machine scheduling problem in which an availability constraint is imposed on one machine as well as on both machines [15]. The problem was optimally solved by Lee using pseudopolynomial dynamic programming algorithms. Mosheiov addressed the scheduling issue in the context of identical parallel machines with availability constraints [16]. Sadfi and Ouarda studied a dynamic programming approach to solving the parallel machine scheduling problem with availability constraints [21]. More scheduling problems where machines are not continuously available for processing can be found in [22]. Although the above schemes considered scheduling problems with availability constraints, they are inadequate for multiclass applications running in heterogeneous systems because they either focused on a single machine [1], [13], [18], two machines [15], or a homogeneous system [16], [21], [22]. Besides, all of them only considered applications with one single-class task. To remedy this issue, in this paper, we address the problem of scheduling multiple classes of tasks with availability constraints in heterogeneous systems. Specifically, we aim to develop a novel scheduling strategy used to enhance the availability of heterogeneous systems while maintaining good performance in the average response time of multiclass tasks. In this study, we consider

Poisson arrivals, in which case we design an availability-aware scheduling algorithm applied to heterogeneous systems where the computing capacity and availability constraints are known a priori.

In our previous work, we studied security-aware scheduling for embedded systems [32], clusters [31], [33], and Grids [34]. However, these scheduling algorithms are designed for homogeneous systems. Further, our previous scheduling algorithms are not suitable for multiclass tasks with availability requirements. In contrast, the algorithm proposed in this paper makes a good trade-off between availability and responsiveness. The rest of the paper is organized as follows: Section 2 describes a system model of heterogeneous systems with availability constraints. Section 3 presents a scheduling algorithm focused on improving the availability of applications in heterogeneous computing environments. Section 4 is devoted to evaluating the performance of the proposed scheduling algorithm. We conclude the paper with future work in Section 5.

2 MODEL DESCRIPTION AND PROBLEM FORMULATION

2.1 Architecture Model

We consider a queuing architecture of a heterogeneous system in which n nodes are connected via a network to process independent m classes of nonpreemptive tasks submitted by m users. Both m and n are finite integers that are greater than or equal to 1. Let $N = \{n_1, n_2, \dots, n_j, \dots, n_n\}$ denote the set of heterogeneous nodes. We assume that the nodes differ only in their speeds and availability levels (hereinafter, the terms "availability level of a node" and "availability of a node" are used interchangeably). The system architecture model, depicted in Fig. 1, is composed of a task schedule queue, a Scheduling Strategy for multiple classes of tasks with Availability Constraints (SSAC) task scheduler, and n local task queues. The goal of SSAC is to make a good task allocation decision for each class of tasks to satisfy their availability requirements and maintain an ideal performance in response time.

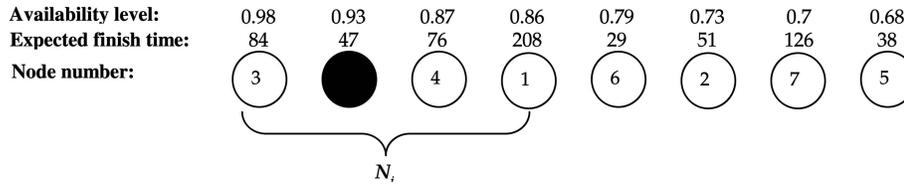


Fig. 2. Example node list sorted by availability level.

A schedule queue is used to accommodate incoming tasks. The SSAC scheduler then processes all arrival tasks in a *first-come, first-served (FCFS)* manner. After being handled by SSAC, the tasks are dispatched to one of the designated nodes $n_j \in N$ for execution. The nodes, each of which maintains a local queue, can execute tasks in parallel. The centerpiece of the system architecture model is SSAC, which is composed of three modules: 1) the Availability Provider Locator (APL), 2) the Availability Cost Calculator (ACC), and 3) the Load Imbalance Detector (LID). For tasks of each class, the APL is used to find all nodes that can meet the tasks' availability requirements and put these nodes in the set N_i . If N_i is nonempty, the APL will choose a node in N_i that can offer tasks of the class with the minimal expected response time as a candidate node. An empty N_i indicates that no node in the system can meet the availability constraints of the current task class. In this case, SSAC will employ the ACC to calculate the availability cost of each node in N for the current class. The node with the least availability cost will be selected as a candidate node. The function of LID is to detect whether or not the candidate node is overloaded. If it is overloaded, the current task class will be assigned to the node with the lightest load. Otherwise, the task class will be allocated to the candidate node. A detailed description of the SSAC strategy can be found in Section 3. To illustrate how APL works, we give an example (see Fig. 2).

In Fig. 2, we assume that there are eight nodes in a system. The first row shows the availability levels that the eight nodes exhibit. The second row displays the expected finish times for task class i on the nodes. The third row is a node list sorted by the node's availability level in a nondecreasing order. We suppose that the task's availability requirement is 0.85. Therefore, the first four nodes (3, 8, 4, 1) will be put into set N_i as all of them can fully satisfy the task's availability requirement. SSAC will eventually choose node 8 (the black node) as the candidate node because it can minimize the expected response time of the task class.

2.2 Modeling Multiclass Tasks with Availability Requirements

For future reference, we summarize the notation that is used throughout this paper in Table 1. There are m classes of tasks submitted to a heterogeneous system by users. Tasks are independent of one another. Each class of tasks requires a common availability specified by a user. Values of availability levels are normalized in the range from 0 to 1.0. For example, users may set the availability levels of critical task classes to 1.0, which means that critical tasks should be assigned to a node that ensures that the task can be successfully completed.

Since arrival patterns and service rates can be estimated by code profiling and statistical prediction [5], it is assumed in this study that the arrival patterns and service rate is known a priori. Without loss of generality, we assume that the tasks of the i th ($1 \leq i \leq m$) class arrive according to a Poisson process with rate λ_i . All classes of tasks arrive at the system at an aggregate rate of $\lambda = \sum_{i=1}^m \lambda_i$. Let p_{ij} be the probability that the tasks of the i th class are dispatched to node j , where $1 \leq j \leq n$. Then, the aggregate task arrival rate of the j th node is expressed as

$$\Lambda_j = \sum_{i=1}^m p_{ij} \lambda_i. \quad (1)$$

Let μ_{ij} denote the service rate of tasks in class i allocated on node j and the corresponding expected service time is computed by $1/\mu_{ij}$. It should be noted that the service time of the i task class on node j has a general distribution, which is independent of the arrival processes. Thus, the service utilization of class i can be written as

TABLE 1
Definitions of Notation

Notation	Definition
n	Number of nodes in a heterogeneous system. ($1 \leq n < \infty$)
m	Number of task classes submitted to the system. ($1 \leq m < \infty$)
λ_i	Arrival rate of tasks in the i th class.
Λ_j	Aggregate task arrival rate of the j th node. (see Eq. 1)
p_{ij}	Probability that tasks of the i th class are dispatched to node j .
μ_{ij}	Service rate of tasks in class i allocated on node j .
ρ_i	Service utilization of class i . (see Eq. 2)
ϕ_j	Service utilization for all tasks allocated to node j . (see Eq. 3)
ϕ	Summation of the service utilizations of all nodes. (see Eq. 4)
TN_j	Average response time of node j . (see Eq. 5)
TC_i	Expected response time of class i tasks. (see Eq. 8)
T	Mean response time of jobs averaged over all the classes. (see Eq. 9)
ξ_j	Availability of node j . ($0 \leq \xi_j \leq 1$)
a_i	Availability requirement of class i . ($0 \leq a_i \leq 1$)
δ_j	Availability shortage of node j . (see Eq. 11)
d_{ij}	Availability shortage factor of class i on node j . (see Eq. 12)
AC_i	Availability cost of class i on node j . (see Eq. 14)
θ_j	Unavailable rate of node j . (see Eq. 15)
AC_i	Availability cost of class i . (see Eq. 16)
A_i	Availability of class i . (see Eq. 17)
A	Availability exhibited by the system. (see Eq. 18)

$$\rho_i = \sum_{j=1}^n (p_{ij}\lambda_i/\mu_{ij}). \quad (2)$$

Similarly, we can obtain the service utilization for all tasks allocated to node j as follows:

$$\phi_j = \sum_{i=1}^m (p_{ij}\lambda_i/\mu_{ij}). \quad (3)$$

The total service utilization of a heterogeneous system, which can be derived from (3), is the summation of the service utilizations of all the nodes. Thus, we have

$$\phi = \sum_{j=1}^n \phi_j = \sum_{j=1}^n \sum_{i=1}^m (p_{ij}\lambda_i/\mu_{ij}). \quad (4)$$

In this study, each node in the system is modeled as a single M/G/1 queue. Thus, the average response time of node j can be computed as

$$TN_j = E(s_j) + \frac{\Lambda_j E(s_j^2)}{2(1-\phi_j)}, \quad (5)$$

where $E(s_j)$ and $E(s_j^2)$ are the mean and mean-square service times. $E(s_j)$ and $E(s_j^2)$ are given as

$$E(s_j) = \sum_{i=1}^m \left(\frac{p_{ij}\lambda_i}{\Lambda_j} \cdot \frac{1}{\mu_{ij}} \right) = \frac{1}{\Lambda_j} \sum_{i=1}^m \left(\frac{p_{ij}\lambda_i}{\mu_{ij}} \right), \quad (6)$$

$$E(s_j^2) = \sum_{i=1}^m \left(\frac{p_{ij}\lambda_i}{\Lambda_j} \cdot s_{ij}^2 \right) = \frac{1}{\Lambda_j} \sum_{i=1}^m (p_{ij}\lambda_i s_{ij}^2), \quad (7)$$

where s_j is the service time of multiple task classes on node j , s_j^2 is the second moments of the service time, and s_{ij}^2 is the second moments of the service time experienced by the tasks of the i th class on node j .

The expected response time TC_i of the i th class tasks can be readily derived from the average response times of the nodes (see (5)). Hence, we obtain TC_i as given by (8):

$$TC_i = \sum_{j=1}^n (p_{ij} \cdot TN_j). \quad (8)$$

Now, we derive the mean response time of jobs averaged over all the classes from (8) as

$$\begin{aligned} T &= \sum_{i=1}^m \left(\frac{\lambda_i}{\lambda} TC_i \right) \\ &= \sum_{i=1}^m \left(\frac{\lambda_i}{\lambda} \sum_{j=1}^n (p_{ij} \cdot TN_j) \right). \end{aligned} \quad (9)$$

To minimize the average response time without taking availability constraints into account, we have to balance the load of the nodes by evenly distributing the service utilization. In other words, making all of the node service utilization equal leads to a perfect load balance. Therefore, we have

$$\phi_j = \sum_{i=1}^m (p_{ij}\lambda_i/\mu_{ij}) = \phi_0. \quad (10)$$

Now, we are positioned to consider availability constraints in the context of heterogeneous computing environments. Formally, instantaneous availability of a system is the probability that the system is not only performing properly without failures, but also satisfying the specified performance requirements [12]. Steady-state availability is the probability that a system is running during any period in which the system is required to be operational [12]. For simplicity and without loss of generality, we refer to steady-state availability as availability throughout this paper. Thus, the availability of node j is characterized by the probability ξ_j that the node is continuously operational for computation during any random period. The availability of a node is modeled as a function determined by a variety of factors, including the node's maintenance status, the number of spare devices dedicated for the node, and the presence or absence of antivirus software. To determine the value of θ_j of node j , we used the fuzzy-logic-based trust model proposed in [28] to aggregate the multiple factors into a normalized scalar value. Detailed information regarding the trust model can be found in [28].

Although the availability of a node could be a dynamically changed value in the long term due to periodic maintenance, regular upgrades, and sudden invasions of malicious codes, it can be approximated to a constant value during a short period of time like a complete execution cycle of a multiclass application. In other words, the availability of a node is independent of the allocations of task classes, meaning that allocating class i to node j has no impact on the availability of node j for other classes.

We denote a_i as the availability requirement of task class i . Specifically, a_i is the probability that the tasks of class i must be successfully executed. Different classes of tasks have distinct availability requirements determined by the consequences of failures of their executions. Failures of critical tasks are catastrophic, whereas failures of noncritical tasks are relatively less damaging. As a result, a critical task requires being assigned to a node with high availability because failures of the task could be catastrophic. A noncritical task may be assigned to a node with a medium availability level because failures of the task will not lead to severe consequences. For example, the availability requirement of a critical task might be 0.95, meaning that the possibility that the task fails must not be higher than 0.05. It should be noted that a_i and λ_i are mutually independent of each other. We quantify the availability of a heterogeneous system by introducing the concepts of *availability shortage factor* and *availability shortage*, which characterize the discrepancy between availability demands and the actual offered availability. The availability shortage factor d_{ij} of task class i on node j is modeled as a step function. Thus, we have

$$d_{ij} = \begin{cases} 0, & \text{if } a_i \leq \xi_j \\ a_i - \xi_j, & \text{otherwise, } 0 \leq a_i \leq \xi_j \leq 1. \end{cases} \quad (11)$$

The availability shortage of node j is calculated based on the availability factor as

$$\delta_j = \sum_{i=1}^m \frac{p_{ij}\lambda_i}{\Lambda_j d_{ij}} = \frac{1}{\Lambda_j} \sum_{i=1}^m p_{ij}\lambda_i d_{ij}, \quad (12)$$

where $\Lambda_j = \sum_{i=1}^m p_{ij}\lambda_i$.

The availability shortage of the system is written as the accumulative sum of availability shortages of all the nodes. Thus, we have

$$\delta = \sum_{j=1}^n \delta_j = \sum_{j=1}^n \sum_{i=1}^m \frac{p_{ij} \lambda_i}{\Lambda_j} d_{ij} = \sum_{j=1}^n \frac{1}{\Lambda_j} \sum_{i=1}^m p_{ij} \lambda_i d_{ij}. \quad (13)$$

Equation (13) measures the discrepancy between the system availability and the availability requirements imposed by tasks. We can make use of the concept of availability shortage to measure satisfaction degrees in terms of availability. However, it is inadequate to leverage the availability shortage to quantitatively evaluate system availability for all of the classes of tasks during their executions on the system.

To remedy this situation, we model the availability of the system for all of the classes as below. Our availability model is motivated by the reliability models found in the literature [25], [29]. Since the availability model relies on the concept of availability cost, let us first introduce the availability cost of class i on node j using

$$AC_{ij} = p_{ij} \frac{\theta_j}{\mu_{ij}}, \quad (14)$$

where θ_j is the unavailable rate of node j .

Equation (14) shows that the availability cost of class i on node j is directly proportional to two parameters: 1) the probability that tasks of the i th class are dispatched to node j and 2) the unavailable rate of node j . Note that the unavailable rate used in this study is expressed as (15), where α is a system parameter. The value of α used in our experiments is 0.1. Equation (15) indicates that the unavailable rate of node j is inversely proportional to the availability of node j . System parameter α must agree with the measurements taken from real systems, whereas availability ξ_j can be estimated and provided by hardware vendors. It is worth noting that this way of calculating unavailable rates is only for illustration purposes and it is flexible to substitute any unavailable rate model for

$$\theta_j = 1 - \exp(-\alpha(1 - \xi_j)). \quad (15)$$

The availability cost A_i of class i is derived from (14) and (15) as follows:

$$AC_i = \sum_{j=1}^n AC_{ij} = \sum_{j=1}^n p_{ij} \frac{\theta_j}{\mu_{ij}}. \quad (16)$$

Based on (16), we can express the availability A_i experienced by class i as (17). Note that this availability model is very similar to some reliability models proposed in the literature [25], [29]:

$$A_i = \exp[-AC_i] = \exp\left[-\sum_{j=1}^n p_{ij} \frac{\theta_j}{\mu_{ij}}\right]. \quad (17)$$

Now, we calculate the availability A exhibited by the system. The system's availability expressed by (18) is the probability that the system is continuously performing at any random period of time. Alternatively, the system's availability can be computed by the expected fraction of

time in which the system is performing during the period in which it is required to be operational [12]:

$$A = \sum_{i=1}^m \left(\frac{\lambda_i}{\lambda} A_i \right) = \sum_{i=1}^m \left\{ \frac{\lambda_i}{\lambda} \exp\left[-\sum_{j=1}^n \left(p_{ij} \frac{\theta_j}{\mu_{ij}} \right) \right] \right\}. \quad (18)$$

Equation (18) indicates that, to enhance the system availability, we must substantially reduce the availability cost expressed by (16).

2.3 Problem Formulation

Now, we formulate the scheduling problem as a trade-off problem between availability and the mean response time. Thus, the proposed scheduling algorithm aims at improving system availability (see (18)) and maintaining an ideal response time of submitted tasks (see (9)). More formally, the problem of maximizing the availability of a heterogeneous system can be formulated as follows:

$$\text{Maximize } A = \sum_{i=1}^m \left\{ \frac{\lambda_i}{\lambda} \exp\left[-\sum_{j=1}^n \left(p_{ij} \frac{\theta_j}{\mu_{ij}} \right) \right] \right\}$$

subject to the following response time constraints:

$$\forall 1 \leq i \leq m, 1 \leq j \leq n, a_i \leq \xi_j : \text{Minimize } TC_i.$$

The above constraint is the response time constraints, each of which means that, among nodes whose availability shortage factor for class i equals zero, a node is chosen for the i th class in such a way as to minimize the mean response time of class i . Notice that the response time constraints can be satisfied by estimating the mean response times of class i on all candidate nodes whose availability shortage factor for class i is zero.

2.4 Heterogeneity Model

Each node in the architecture model (Fig. 1) is inherently heterogeneous in both computational speed and availability level. Computational heterogeneity captures the nature of heterogeneous computing platforms where the execution times of each task on different nodes are distinctive. Although each multiclass task has an availability requirement, computational nodes exhibit a variety of availability levels. For simplicity and without loss of generality, the availability levels and availability requirements are normalized in the range from 0 to 1.0.

We introduce the concepts of computational heterogeneity and availability heterogeneity. The computational weight of class i on node j is defined as the ratio between its service rate on node j and the fastest service rate in the system. That is, the computational weight is expressed by $w_{ij} = \mu_{ij} / \max_{k=1}^n (\mu_{ik})$.

The computational heterogeneity of the i th class, that is, HC_i , can be measured by the standard deviation of the computational weights. Thus, we have

$$HC_i = \sqrt{\frac{1}{n} \sum_{j=1}^n (\bar{w}_i - w_{ij})^2}, \quad (19)$$

where \bar{w}_i is the average computational weight, that is, $\bar{w}_i = (\sum_{j=1}^n w_{ij})/n$.

The computational heterogeneity can be expressed as follows:

$$HC = \frac{1}{m} \sum_{i=1}^m HC_i. \quad (20)$$

The heterogeneity of availability HA in a heterogeneous system is measured by the standard deviation of the availability offered by all the nodes in the system. Hence, HA is written as

$$HA = \sqrt{\frac{1}{n} \sum_{j=1}^n (\bar{\xi} - \xi_j)^2}, \text{ where } \bar{\xi} = \left(\sum_{j=1}^n \xi_j \right) / n. \quad (21)$$

2.5 Load Imbalance Detection Mechanism

To fully satisfy the tasks' availability requirements, SSAC tends to assign tasks to a group of nodes (a subset of N) that can provide high availability levels. Note that the availability level offered by a node is orthogonal to its computational speed. The implication is that SSAC might assign a large number of tasks onto a node with a high availability level and low computational speed. As a result, the mean response time achieved by SSAC could suffer significantly due to load imbalance. To prevent severe load imbalance from occurring, SSAC leverages a load imbalance detection mechanism called LID to detect whether or not a node j in the system is overloaded. LID uses load index L_j , defined as follows to measure relative workload of node j :

$$L_j = \frac{\phi_j}{\phi/n}, \quad (22)$$

where ϕ_j is the service utilization of node j (see (3)), and ϕ/n is the average node service utilization of the whole system (see (4)). When L_j is higher than a threshold value TL , node j is overloaded. Note that TL is an empirical parameter and we set it to 1.25 in our experiments. The service utilization of node j is essentially the traffic intensity of node j .

3 THE AVAILABILITY-AWARE SCHEDULING ALGORITHM

In this section, we present SSAC.

We now present the SSAC scheduling strategy, which is intended to determine probability $\{p_{ij}\}_{1 \leq i \leq m, 1 \leq j \leq n}$ in a judicious way to improve the availability of heterogeneous systems while maintaining good performance in response time. Since the average response time largely depends on the sequencing strategies used in each node, we employ an existing optimal sequencing strategy [24], [27] to minimize the average response time of all classes (see (9)). Our approach relies on the following proposition that can be proved based on Proposition 2.1 in [24].

Proposition 1. *Given an m -class $M/G/1$ queue and an n -node heterogeneous system, class i has arrival rate λ_i and service rate μ_{ij} on node j . The scheduling policy on node j that gives*

```

1. Sort and label classes such that  $\lambda_1 / \sum_{j=1}^n \mu_{1j} \geq \lambda_2 / \sum_{j=1}^n \mu_{2j} \cdots \geq \lambda_m / \sum_{j=1}^n \mu_{mj}$ ;
2. for each class  $i$  do
3. Initialize the availability cost and response time for class  $i$ , i.e.,  $AC \leftarrow \infty, TC \leftarrow \infty$ ;
4. Create a set  $N_i$  of nodes, where node  $j \in N_i$  if  $a_i \leq \xi_j$ ;
5. if  $(N_i \neq \emptyset)$  then
6. for each node  $j$  in  $N_i$  do
7.  $\rho_{ij} \leftarrow 1$ ; calculate expected response time of class  $i$ ,  $TC_i$ ; (see Eq. 8)
8. if  $TC_i < TC$  then
9.  $TC \leftarrow TC_i; v \leftarrow j$ ;
10. end for
11. else
12. for each node  $j$  in the system do
13. Calculate the availability cost of class  $i$  on node  $j$ ,  $AC_{ij}$ ; (see Eq. 14)
14. if  $AC_{ij} < AC$  or ( $AC_{ij} = AC$  and  $TC_i < TC$ ) then
15.  $AC = AC_{ij}; TC \leftarrow TC_i; v \leftarrow j$ ;
16. end for
17. end if
18.  $n_{min} = 1; L_{min} = \infty$ ; /* Assume node 1 is the lightest load node and its load index is  $\infty$  */
19. for each node  $n_j \in N$  do
20. Calculate its load index  $L_j$ ; (see Eq. 22)
21. if  $L_j < L_{min}$  then
22.  $L_{min} = L_j; n_{min} = j$ ;
23. end for
24. if  $L_v \leq TL$  then /* node  $v$  is not overloaded */
25.  $\rho_{iv} \leftarrow 1$ ; /* indicate that class  $i$  to node  $v$  */
26. Allocate class  $i$  to node  $v$ ;
27. else
28. Allocate class  $i$  to the node  $n_{min}$ ;
29. end if
30. end for

```

Fig. 3. The scheduling strategy for multiple classes of tasks with availability constraints.

priority to class i over k whenever $\mu_{ij} \geq \mu_{kj}$ minimizes the expected response time $T = \sum_{i=1}^m (\frac{\lambda_i}{\lambda} TC_i)$ (see (9)).

Proposition 1 indicates that classes with higher service utilization must be given a high priority in the process of scheduling. For simplicity, we have the following assumption.

Assumption 1. *The classes are labeled such that*

$$\lambda_1 / \sum_{j=1}^n \mu_{1j} \geq \lambda_2 / \sum_{j=1}^n \mu_{2j} \cdots \geq \lambda_m / \sum_{j=1}^n \mu_{mj},$$

where $\lambda_i / \sum_{j=1}^n \mu_{ij}$ is the service utilization of task class i .

This assumption is valid because the first step of the algorithm can sort the task classes in such a way before having the task classes relabeled. The relabeling process of the algorithm assigns a number of priority levels to the task classes. That is, the priority of class i is higher than that of k if $i < k$. Based on the standard queuing theory, the expected response time for tasks of class i on node j can be approximated by the following equation:

$$TC_{ij} = W_i + \frac{1}{\mu_{ij}} = \frac{\sum_{i=1}^m p_{ij} \lambda_i E(s_i^2)}{2(1 - \sum_{l < i} \rho_{lj})(1 - \sum_{l \leq i} \rho_{lj})}, \quad (23)$$

where $\rho_{lj} = \frac{p_{lj} \lambda_j}{\mu_{lj}}$.

The SSAC algorithm, which is outlined in Fig. 3, aims at improving availability while achieving low average response time for multiclass tasks running in heterogeneous systems. First, SSAC is intended to give higher priorities to classes with higher service utilization (see Assumption 1). To achieve this goal, SSAC sorts and relabels all of the classes in such a way that the condition $\lambda_1 / \sum_{j=1}^n \mu_{1j} \geq \lambda_2 / \sum_{j=1}^n \mu_{2j} \cdots \geq \lambda_m / \sum_{j=1}^n \mu_{mj}$ is satisfied (see Step 1).

Step 3 sets the initial values of the availability cost and response time for task class i to infinity. Step 4 determines a

set N_i of nodes that can meet the availability demands of tasks of class i . Specifically, a node $j \in N_i$ meets the availability constraints of class i if the i th class's availability requirement is smaller than the availability level offered by N_i (that is, $a_i \leq \xi_j$).

Steps 5-17 are at the core of the SSAC algorithm. Step 5 determines if there exists at least one node whose availability shortage factor for class i equals zero. This process is implemented in Step 5 by checking if set N_i has at least one element. In the case where there is one or more nodes with zero availability shortage factor for class i , Steps 6-10 aim at reducing the mean response time of class i by estimating the mean response times of class i on all candidate nodes in set N_i . Thus, the SSAC algorithm chooses the most appropriate node from set N_i in a such way that the selected candidate node can provide task class j with the minimal response time estimated by Step 7.

There is a possibility that node set N_i is empty, meaning that no node in the heterogeneous system is capable of guaranteeing the availability constraint of class i . In this case, Steps 12-16 make an effort to improve system availability derived from (18). More specifically, Step 13 leverages (14) to compute the availability cost AC_{ij} of class i on node j . Then, Steps 14 and 15 gradually reduce the availability cost value AC , thereby enhancing the system availability characterized by (18). If two nodes offer the same availability for class i , the node offering a smaller mean response time will be chosen for class i to break the tie (see Step 14).

Steps 18-23 calculate the load index value L_j for each node j in the system and find a node n_{min} with the lightest load L_{min} . If node v is not overloaded ($L_v \leq TL$), Steps 25 and 26 allocate tasks of class i to node v , which is expected to enhance the system availability.

The mean response time of all of the classes is further reduced through static load balancing (see Step 28). Specifically, in case node v is overloaded, Step 28 allocates class i to a node with the lightest load. Node i is considered overloaded if its load index is greater than TL , which is set to 1.25 in our experiments.

To analyze the computational overhead of SSAC, we obtain its worst time complexity as follows:

Theorem 1. *The worst-case time complexity of SSAC is $O(m(\log m + 2n + 1))$, where n is the number of nodes and m is the number of classes.*

Proof. The time complexity of sorting and labeling multiple classes is $O(m \log m)$ (Step 1). If $N_i = \emptyset$, it takes $O(n)$ time to maximize availability by reducing the availability cost (Steps 12-16) and Steps 6-10 will be skipped. If $N_i = N$, it takes $O(n)$ time to discover a node who can offer the minimal expected time for the current class of tasks (Steps 6-10) and Steps 12-16 will be skipped. In case $N_i \subset N$ and $N_i \neq \emptyset$, it takes $O(k)$ (k is the length of N_i , and $1 \leq k < n$) time to discover a node that offers the minimal expected response time for the current class of tasks (Steps 6-10) and Steps 12-16 will be ignored. Therefore, the worst case for Steps 5-17 is $O(n)$. Additionally, it takes $O(n)$ time to calculate the load index of each node in the system (Steps 19-23). For other steps, they only consume $O(1)$. Since the total number of

classes is m , the time complexity for the process of optimizing availability (Steps 2-30) is $O((n + 1)m)$. Thus, the worst time complexity of the SSAC algorithm is $O(m \log m) + O((2n + 1)m) = O(m(\log m + 2n + 1))$. \square

Since m and n are all finite integers, which are not big numbers in practice, Theorem 1 shows that the time complexity of SSAC is low in most cases. This time complexity indicates that the execution time of SSAC is a small value compared with task execution times. Thus, the CPU overhead of executing SSAC is ignored in our experiments.

The following two theorems show important features of the SSAC strategy. Assuming that all nodes in a heterogeneous system are able to fulfill the availability requirements of all task classes, we can prove the following two theorems regarding the availability shortage and mean response time of the system. Theorem 2 demonstrates that, if each node $j \in N$ can fully satisfy the availability requirements of tasks of any class i , there is no availability shortage in node j (availability provider) for tasks of class i (availability consumer). Theorem 2 implies that, in this perfect availability satisfaction scenario, minimizing the mean response time of the system becomes the only goal pursued by SSAC.

Theorem 2. *In a workload where the maximal availability requirement among all classes is less than or equal to the minimal availability among all nodes in a system, then the availability shortage of the system is zero. Thus, $\max_{i=1}^m(a_i) \leq \min_{j=1}^n(\xi_j) \rightarrow \delta = 0$.*

Proof. Given a task class k ($1 \leq k \leq m$) and a node l ($1 \leq l \leq n$), we have $a_k \leq \max_{i=1}^m(a_i) \leq \min_{j=1}^n(\xi_j) \leq \xi_l$. Since $a_k \leq \xi_l$, it follows that, $\forall 1 \leq k \leq m$,

$$1 \leq l \leq n : d_{kl} = 0.$$

Therefore, we obtain

$$\begin{aligned} \delta &= \sum_{j=1}^n \delta_j = \sum_{j=1}^n \sum_{i=1}^m \frac{p_{ij} \lambda_i}{\Lambda_j} \cdot d_{ij} \\ &= \sum_{j=1}^n \frac{1}{\Lambda_j} \sum_{i=1}^m p_{ij} \cdot \lambda_i \cdot 0 = 0, \end{aligned}$$

which completes the proof of Theorem 2. \square

Theorem 3. *In a workload where the maximal availability requirement among all classes is less than or equal to the minimal availability among all nodes in a system (that is, $\max_{i=1}^m(a_i) \leq \min_{j=1}^n(\xi_j)$), then the mean response time of the system is*

$$\sum_{i=1}^m \frac{\lambda_i}{\lambda} \sum_{j=1}^n \left(p_{ij} \cdot \left(\frac{\phi_0}{\Lambda_j} + \frac{\sum_{k=1}^m (p_{kj} \lambda_k s_{kj}^2)}{2(1 - \phi_0)} \right) \right),$$

where $\phi_0 = \phi_j = \sum_{i=1}^m (p_{ij} \lambda_i / \mu_{ij})$, $1 \leq j \leq n$.

Proof. Since $\max_{i=1}^m(a_i) \leq \min_{j=1}^n(\xi_j)$ means that all of the nodes can meet the availability requirements of all the task classes, Steps 6-10 and 24-29 in the SSAC algorithm judiciously reduce the mean response time by balancing

the load of the nodes. Hence, $\phi_j = \sum_{i=1}^m (p_{ij}\lambda_i/\mu_{ij}) = \phi_0$, $1 \leq j \leq n$. It follows that

$$\begin{aligned} T &= \sum_{i=1}^m \left(\frac{\lambda_i}{\lambda} TC_i \right) = \sum_{i=1}^m \left(\frac{\lambda_i}{\lambda} \sum_{j=1}^n (p_{ij} \cdot TN_j) \right) \\ &= \sum_{i=1}^m \frac{\lambda_i}{\lambda} \sum_{j=1}^n \left(p_{ij} \cdot \left(E(s_j) + \frac{\Lambda_j E(s_j^2)}{2(1-\phi_o)} \right) \right) \\ &= \sum_{i=1}^m \frac{\lambda_i}{\lambda} \sum_{j=1}^n \left(p_{ij} \cdot \left(\frac{\phi_o}{\Lambda_j} + \frac{\sum_{k=1}^m (p_{kj} \cdot \lambda_k \cdot s_{kj}^2)}{2(1-\phi_o)} \right) \right). \end{aligned}$$

We therefore conclude that the mean response time of the system can be calculated as

$$\sum_{i=1}^m \frac{\lambda_i}{\lambda} \sum_{j=1}^n \left(p_{ij} \cdot \left(\frac{\phi_o}{\Lambda_j} + \frac{\sum_{k=1}^m (p_{kj} \cdot \lambda_k \cdot s_{kj}^2)}{2(1-\phi_o)} \right) \right)$$

and the proof of Theorem 3 is complete. \square

Since homogeneous systems widely deployed in the real world are a special case of heterogeneous systems, we study the behaviors of SSAC in a homogeneous system in the following two theorems. Theorems 4 and 5 below capture the characteristic behaviors of SSAC in the context of homogeneous systems. Specifically, Theorem 4 establishes that, in the case where all nodes in a homogeneous system are incapable of guaranteeing the availability requirements of a task class, the SSAC algorithm initially allocates the class to a node with the highest availability among all of the nodes in the system. Thus, we have the following theorem:

Theorem 4. *Given a task class i whose availability requirement cannot be satisfied by any node in a homogeneous system with n nodes (that is, $a_i > \max_{k=1}^n (\xi_k)$), SSAC initially allocates class i to a node j whose availability is the highest among the n nodes, that is, $\xi_j = \max_{k=1}^n (\xi_k)$. Formally, if*

$$\forall 1 \leq i \leq m : (\forall 1 \leq j, k \leq n, j \neq k : \mu_{ij} = \mu_{ik} = \mu_i),$$

then $a_i > \max_{k=1}^n (\xi_k) \rightarrow$ SSAC initially allocates class i to node j subject to $\xi_j = \max_{k=1}^n (\xi_k)$.

Proof. Since we have $a_i > \max_{j=1}^n (\xi_j)$, it is intuitive that all of the nodes in the homogeneous system are unable to meet the availability requirement of task class i . Thus, we show that the set N_i of nodes for class i is empty (see Step 4 in Fig. 3). In this case, Steps 12-16 are executed to determine a node j that offers the smallest availability cost among all of the nodes in the system (see Step 14 in Fig. 3). The initial availability cost of class i on node j is equal to θ_j/μ_{ij} (see (14)). Because the n nodes are homogeneous (that is, $\mu_{ij} = \mu_i$), the availability cost of class i on node j can be expressed as θ_j/μ_i . If the value of θ_j/μ_i is the smallest among all of the nodes and μ_i is a constant, then the value of unavailable rate θ_j is also the smallest. The smallest value of θ_j indicates the highest availability ξ_j of node j among all of the n nodes. Hence, we show that SSAC initially allocates class i to a node j whose availability is the highest among the n nodes. \square

Theorem 5 states that, if all nodes in a homogeneous system are incapable of guaranteeing the availability requirements of all task classes (that is, $\max_{j=1}^n (\xi_j) < \min_{i=1}^m (a_i)$), the SSAC strategy gracefully degrades to a load-balancing scheme.

Theorem 5. *If all of the nodes in a homogeneous system are incapable of guaranteeing the availability requirements of all task classes (that is, $\max_{j=1}^n (\xi_j) < \min_{i=1}^m (a_i)$), the SSAC strategy gracefully degrades to a load-balancing scheme.*

Proof. Given $\max_{j=1}^n (\xi_j) < \min_{i=1}^m (a_i)$, we show that $\forall 1 \leq i \leq m, 1 \leq j \leq n : \xi_j < a_i$. This means that, for any task class i , no node in the system can guarantee the class's availability requirement. Hence, the set N_i of nodes for class i is empty (see Step 4 in Fig. 3), making SSAC allocate class i to a node j offering the smallest availability cost among all of the nodes in the system (see Step 14 in Fig. 3). Based on Theorem 4, it is proven that Steps 12-16 in SSAC initially attempt to allocate all of the classes to a set N_{HA} of nodes whose availability is the highest among the n nodes. Therefore, nodes in set N_{HA} have a high likelihood of being overloaded. If any node in set N_{HA} is overloaded, Step 28 of SSAC is invoked to balance the load across all the nodes in the system. Thus, in case all nodes in a homogeneous system are incapable of guaranteeing the availability requirements of all task classes, the SSAC strategy gracefully degrades to a load-balancing scheme. Hence, the proof. \square

4 EXPERIMENTAL RESULTS

We evaluate in this section the performance of the SSAC algorithm using simulation experiments. There are two important workload parameters: the mean arrival rate λ of multiclass tasks and the mean execution time (see Table 1). The system parameters in our experiments are chosen either based on those used in the literature [35] or to represent real-world heterogeneous systems. It is assumed that task arrival times abide by Poisson distribution and task execution times follow a uniform distribution. We evaluated the proposed SSAC algorithm under a wide range of system workloads by varying λ and the number of nodes n . To simulate a heterogeneous system, we randomly generated a vector of n (the number of nodes) execution times for each task using the heterogeneity model described in Section 3. For each simulated result, we performed 1,000 runs, of which the average value is computed after discarding the 10 largest and 10 smallest measurements.

We compared SSAC with two well-known scheduling algorithms to reveal the strengths of the proposed scheduling strategy. The alternative scheduling algorithms are *MINMIN* and *SUFFERAGE* [28], which are nonpreemptive task scheduling algorithms. *MINMIN* and *SUFFERAGE* were selected for comparison purposes because these two algorithms represent many existing algorithms that are closest to our SSAC algorithm. *MINMIN* and *SUFFERAGE* can be applied to allocate a stream of independent tasks to a heterogeneous system. It is important to note that the two alternatives are representative dynamic scheduling algorithms for distributed systems that are either homogeneous or heterogeneous in nature. *MINMIN* and *SUFFERAGE*

TABLE 2
Characteristics of System Parameters

Parameter	Value (Fixed) - (Varied)
Number of nodes	(16) – (16, 32, 64, 128)
Mean task arrival rate λ (Poisson dist.)	(1.0) – (0.2, 0.4, 0.6, 0.8, 1.0)
Task execution time range (Uniform dist.)	(1, 500) second
Node availability (Uniform dist.)	(0.1 – 1.0)
Task availability demands (Uniform dist.)	(0.1 – 1.0)
Computational heterogeneity (computed)	0.35
Availability heterogeneity (computed)	0.22
TL (Threshold value of load index)	1.25

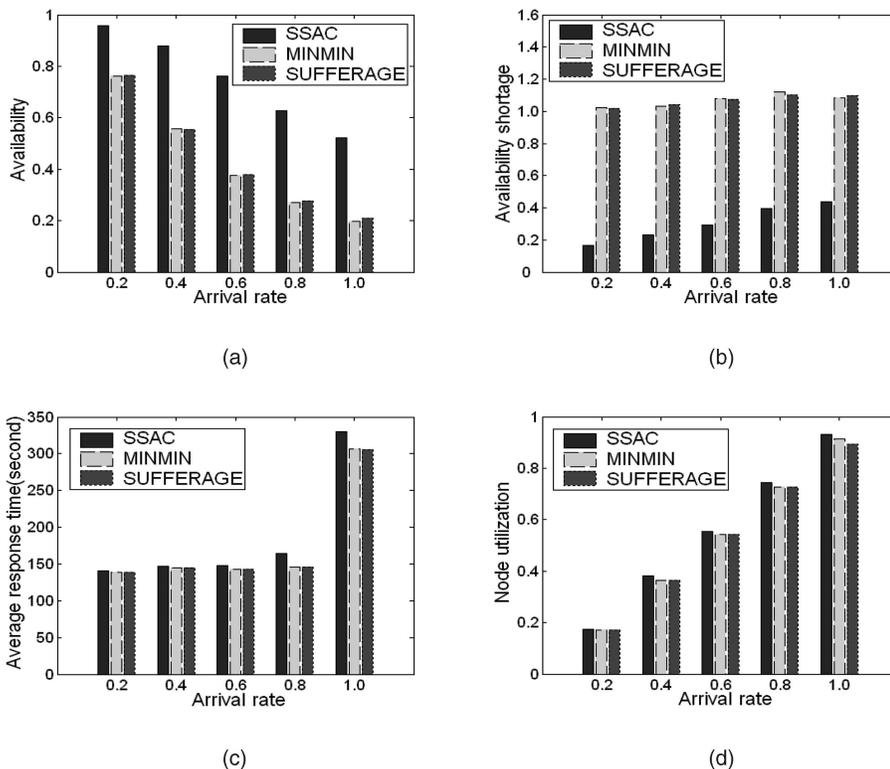


Fig. 4. Performance impact of mean arrival rate λ .

were successfully applied in real-world distributed resources management systems such as SmartNet. These two scheduling algorithms are described in brief as follows:

1. *MINMIN*. For each submitted task, the node providing the earliest completion time is tagged. Among all of the mapped tasks, the one that has the minimal earliest completion time is chosen and then allocated to the tagged node.
2. *SUFFERAGE*. A node is assigned to a task that would “suffer” most in terms of completion time if that node is not allocated to the task.

Table 2 shows the parameters of simulated heterogeneous systems. In what follows, we briefly introduce the performance metrics used to evaluate the performance of the proposed availability-aware scheduling strategy:

1. *Availability* (see (18)). The system’s availability, which is measured by (18), is the probability that

the system is continuously performing at any random period of time.

2. *Availability shortage* (see (13)). The availability shortage of the system quantifies the discrepancy between availability demands and the actual availability offered by the system.
3. *Average response time* (see (9)). The average response time of multiple task classes is the average time interval between the task arrival time and the finish time.
4. *Node utilization*. The utilization of a node is the percentage of total task runtime out of the total available time of the node. *Node utilization* is the average value of all nodes’ utilizations.

In the first group of experiments, we vary the mean arrival rate from 0.2 to 1.0 with an increment of 0.2. Fig. 4 shows experimental results of the three evaluated algorithms applied to a heterogeneous system with 16 nodes. We observe in Fig. 4a that SSAC significantly improves

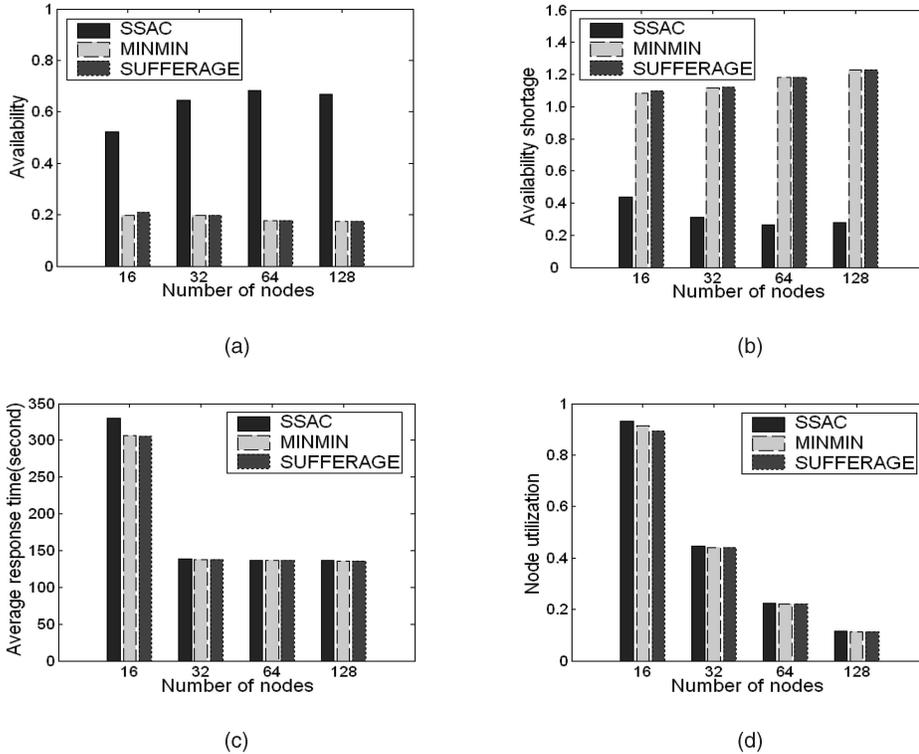


Fig. 5. Performance impact of the number of nodes.

system availability over the two alternatives, whereas the MINMIN and SUFFERAGE algorithms exhibit similar performance in terms of availability. For example, SSAC enhances system availability over the existing approaches by an average of 73.3 percent. We attribute the availability improvements of SSAC over MINMIN and SUFFERAGE to SSAC's capability of considering the tasks' availability requirements in the process of allocating tasks to heterogeneous nodes.

Fig. 4b reveals that the availability shortage of the proposed SSAC is considerably smaller than those of MINMIN and SUFFERAGE. We observe that the results plotted in Fig. 4b are consistent with those reported in Fig. 4a because a low value of availability shortage gives rise to high system availability. Fig. 4c shows that the average response time yielded by SSAC is marginally larger than those generated by MINMIN and SUFFERAGE. More specifically, the performance degradation of our SSAC in terms of response time is less than 5.7 percent on average. In other words, compared with MINMIN and SUFFERAGE, SSAC achieves much better availability performance while maintaining reasonably short response times. Fig. 4d clearly shows that MINMIN and SUFFERAGE perform slightly better than SSAC in terms of node utilization. This is because MINMIN and SUFFERAGE only aim at minimizing response times; therefore, they tend to assign tasks to nodes with high speed while ignoring the tasks' availability requirements. Hence, the average task runtime is relatively shorter. In accordance with the definition of node utilization, a short average task runtime results in low node utilization. Unlike MINMIN and SUFFERAGE, SSAC guarantees the tasks' availability requirements while shortening response times. SSAC may assign tasks to slow nodes

with high availability levels, thereby making tasks have long execution times, which in turn leads to higher node utilization.

An interesting observation drawn from Figs. 4a and 4b is that SSAC outperforms MINMIN and SUFFERAGE in terms of system availability. Furthermore, Fig. 4d reveals that, compared with SSAC, MINMIN and SUFFERAGE can deliver better performance with respect to node utilization. Let us make use of the example in Fig. 2 to explain this phenomenon. As we described in Section 2.1, SSAC selects node 8 (the black one) to meet the task's availability requirement (0.85 in this example). Therefore, $\xi_8 = 0.93$. On the contrary, MINMIN and SUFFERAGE choose node 6 as the candidate node for the tasks of the current class because node 6 can deliver the minimal expected finish time. Thus, $\xi_6 = 0.79$ for MINMIN and SUFFERAGE. Equation (15) indicates that a large value of ξ_j implies a small value of θ_j , which in turn results in a small AC_{ij} . A small value of AC_{ij} gives rise to a high availability A_i (see (17)), which eventually leads to high system availability A (see (18)). In short, a high ξ_j results in high system availability A . Since ξ_8 (selected by SSAC) is noticeably higher than ξ_6 (selected by MINMIN and SUFFERAGE), SSAC outperforms MINMIN and SUFFERAGE in terms of system availability. The rationale is that SSAC judiciously reduces the availability cost by choosing nodes with high availability levels, whereas MINMIN and SUFFERAGE totally ignore the issue of availability cost.

The second group of experiments is focused on the scalability of the SSAC algorithm. In this set of experiments, we vary the number of nodes in the simulated heterogeneous system from 16 to 128. Fig. 5 plots the four

performance metrics of all three examined algorithms as functions of the number of nodes.

An important observation made from Figs. 5a and 5b is that SSAC exhibits good scalability with respect to system availability and availability shortage. This is because Fig. 5a reveals that the performance improvement in system availability becomes more pronounced when the heterogeneous system is scaled up. Similarly, Fig. 5b shows that the availability shortage reduction of SSAC over the two competitive algorithms is more prominent with the increasing number of nodes in the heterogeneous system. Figs. 5a and 5b indicate that the performance gain of SSAC in availability becomes more significant for large-scale heterogeneous systems because a larger number of nodes means a higher probability that SSAC can choose a node to meet each task's availability demands. Figs. 5c and 5d show that, for all of the evaluated scheduling algorithms, the average response time and node utilization reduce as the number of nodes increases. These results are expected because a larger number of nodes implies less work for each node, which in turn leads to a smaller response time on each node.

5 SUMMARY AND FUTURE WORK

An increasing number of applications with availability constraints are running on heterogeneous computing platforms. However, most existing scheduling algorithms in heterogeneous systems ignore the availability requirements imposed by multiclass applications. To remedy this deficiency, we address in this paper the scheduling problem for multiclass applications with availability constraints running in heterogeneous systems. Multiclass tasks are characterized by their execution times and availability requirements, whereas each node in a heterogeneous system is modeled by the node's computing capability and availability. We introduced new metrics to quantify availability and heterogeneity in the context of multiclass tasks. Next, we proposed SSAC, a scheduling algorithm geared to enhance the availability of heterogeneous systems while maintaining good performance in the average response time of multiclass tasks. Empirical results show that, compared with existing schemes, the proposed algorithm significantly improves the availability of multiclass tasks in heterogeneous systems without degrading response times.

As part of future directions, we will extend SSAC to schedule parallel applications with flexible availability requirements. This future work will be accomplished by factoring in communication availability and precedence constraints among tasks. To further improve the performance of the SSAC scheduling algorithm, which is heuristic in nature, in future work, we also plan to explore two efficient algorithms to solve the same problem addressed in this paper. The first algorithm will take an efficient dynamic programming approach, whereas the second algorithm will make use of a branch-and-bound method. A third future direction is to investigate an availability-aware scheduling algorithm that handles cases of Poisson arrivals as well as general arrivals.

ACKNOWLEDGMENTS

The work reported in this paper was supported by the US National Science Foundation under Grants CNS-0713895 and CCF-0742187, the New Mexico Institute of Mining and Technology under Grant 103295, Auburn University under a start-up grant, Intel Corp. under Grant 2005-04-070, and Altera Corp. under an equipment grant. The authors are grateful to the anonymous referees for their insightful suggestions and comments.

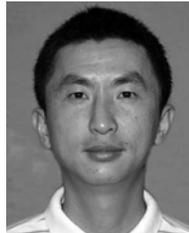
REFERENCES

- [1] I. Adiri, J. Bruno, E. Frostig, and A.H.G. Rinnooy Kan, "Single Machine Flow-Time Scheduling with a Single Breakdown," *Acta Informatica*, vol. 26, pp. 679-696, 1989.
- [2] A. Apon and L. Wilbur, "AmpNet—A Highly Available Cluster Interconnection Network," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS '03)*, Apr. 2003.
- [3] F. Bonomi and A. Kumar, "Adaptive Optimal Load Balancing in a Nonhomogeneous Multiserver System with a Central Job Scheduler," *IEEE Trans. Computers*, vol. 39, no. 10, pp. 199-234, Oct. 1990.
- [4] S.C. Borst, "Optimal Probabilistic Allocation of Customer Types to Servers," *Proc. ACM Conf. Measurement and Modeling Computer Systems (SIGMETRICS '95)*, pp. 116-125, 1995.
- [5] T.D. Braun et al., "A Comparison Study of Static Mapping Heuristics for a Class of Meta-Tasks on Heterogeneous Computing Systems," *Proc. Eighth Heterogeneous Computing Workshop (HCW '99)*, pp. 15-29, Apr. 1999.
- [6] T.L. Casavant and J.G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. Software Eng.*, vol. 14, no. 2, pp. 141-154, Feb. 1988.
- [7] M.E. Crowella, M. Harchol-Balter, and C. Murta, "Task Assignment in Distributed Systems: Improving Performance by Unbalancing Load," *Proc. ACM Conf. Measurement and Modeling Computer Systems (SIGMETRICS '98)*, pp. 268-269, 1998.
- [8] A. Dogan and F. Özgüner, "Reliable Matching and Scheduling of Precedence-Constrained Tasks in Heterogeneous Distributed Computing," *Proc. 29th Int'l Conf. Parallel Processing (ICPP '00)*, pp. 307-314, 2000.
- [9] A. Dogan and F. Özgüner, "LDBS: A Duplication Based Scheduling Algorithm for Heterogeneous Computing Systems," *Proc. 31st Int'l Conf. Parallel Processing (ICPP '02)*, pp. 352-359, 2002.
- [10] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee, "Network Dispatcher: A Connection Router for Scalable Internet Services," *Proc. Seventh Int'l World Wide Web Conf. (WWW '98)*, Apr. 1998.
- [11] Y. Jiang, C.-K. Tham, and C.-C. Ko, "An Approximation for Waiting Time Tail Probabilities in Multiclass Systems," *IEEE Comm. Letters*, vol. 5, no. 4, pp. 175-177, 2001.
- [12] A.M. Johnson and M. Malek, "Survey of Software Tools for Evaluating Reliability, Availability, and Serviceability," *ACM Computing Surveys*, vol. 20, no. 4, pp. 227-269, Dec. 1988.
- [13] I. Kacem, C. Sadfi, and A. El-Kamel, "Branch and Bound and Dynamic Programming to Minimize the Total Completion Times on a Single Machine with Availability Constraints," *Proc. IEEE Int'l Conf. Systems, Man, and Cybernetics (SMC '05)*, vol. 2, pp. 1657-1662, Oct. 2005.
- [14] H.C. Lau and C. Zhang, "Job Scheduling with Unfixed Availability Constraints," *Proc. 35th Ann. Meeting Decision Sciences Inst. (DSI '04)*, pp. 4401-4406, Nov. 2004.
- [15] C.-Y. Lee, "Two-Machine Flowshop Scheduling with Availability Constraints," *European J. Operational Research*, vol. 114, no. 2, pp. 420-429, Apr. 1999.
- [16] G. Mosheiov, "Minimizing the Sum of Job Completion Times on Capacitated Parallel Machines," *Math. and Computer Modelling*, vol. 20, pp. 91-99, 1994.
- [17] D.-T. Peng and K.G. Shin, "Optimal Scheduling of Cooperative Tasks in a Distributed System Using an Enumerative Method," *IEEE Trans. Software Eng.*, vol. 19, no. 3, pp. 253-267, Mar. 1993.
- [18] X. Qi, T. Chen, and F. Tu, "Scheduling the Maintenance on a Single Machine," *J. Operational Research*, vol. 50, pp. 1071-1078, 1999.

- [19] X. Qin and H. Jiang, "A Dynamic and Reliability-Driven Scheduling Algorithm for Parallel Real-Time Jobs on Heterogeneous Clusters," *J. Parallel and Distributed Computing*, vol. 65, no. 8, pp. 885-900, Aug. 2005.
- [20] S. Ranaweera and D.P. Agrawal, "Scheduling of Periodic Time Critical Applications for Pipelined Execution on Heterogeneous Systems," *Proc. 30th Int'l Conf. Parallel Processing (ICPP '01)*, pp. 131-138, Sept. 2001.
- [21] C. Sadfi and Y. Ouarda, "Parallel Machines Scheduling Problem with Availability Constraints," *Proc. Ninth Int'l Workshop Project Management and Scheduling (PMS '04)*, 2004.
- [22] E. Sanlaville and G. Schmidt, "Machine Scheduling with Availability Constraints," *Acta Informatica*, vol. 35, no. 9, pp. 795-811, Sept. 1998.
- [23] G. Schmidt, "Scheduling with Limited Machine Availability," *European J. Operational Research*, vol. 121, pp. 1-15, 2000.
- [24] J. Sethuraman and M.S. Squillante, "Optimal Stochastic Scheduling in Multiclass Parallel Queues," *Proc. ACM SIGMETRICS Conf.*, May 1999.
- [25] S.M. Shatz, J.P. Wang, and M. Goto, "Task Allocation for Maximizing Reliability of Distributed Computer Systems," *IEEE Trans. Computers*, vol. 41, no. 9, pp. 1156-1168, Sept. 1992.
- [26] S.P. Smith, "An Efficient Method to Maintain Resource Availability Information for Scheduling Applications," *Proc. IEEE Int'l Conf. Robotics and Automation (ICRA '92)*, vol. 2, pp. 1214-1219, May 1992.
- [27] W.E. Smith, "Various Optimizers for Single-Stage Production," *Naval Research and Logistics Quarterly*, pp. 59-66, 1954.
- [28] S.-S. Song, K. Hwang, and Y.-K. Kwok, "Risk-Resilient Heuristics and Genetic Algorithms for Security-Assured Grid Job Scheduling," *IEEE Trans. Computers*, vol. 55, no. 6, pp. 703-719, June 2006.
- [29] S. Srinivasan and N.K. Jha, "Safety and Reliability Driven Task Allocation in Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 3, pp. 238-251, Mar. 1999.
- [30] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Trans. Parallel and Distributed Systems*, vol. 13, no. 3, Mar. 2002.
- [31] T. Xie and X. Qin, "Scheduling Security-Critical Real-Time Applications on Clusters," *IEEE Trans. Computers*, vol. 55, no. 7, pp. 864-879, July 2006.
- [32] T. Xie and X. Qin, "Improving Security for Periodic Tasks in Embedded Systems through Scheduling," *ACM Trans. Embedded Computing Systems*, vol. 6, no. 1, 2007.
- [33] T. Xie and X. Qin, "A New Allocation Scheme for Parallel Applications with Deadline and Security Constraints on Clusters," *Proc. IEEE Int'l Conf. Cluster Computing (Cluster '05)*, Sept. 2005.
- [34] T. Xie and X. Qin, "Enhancing Security of Real-Time Applications on Grids through Dynamic Scheduling," *Proc. 11th Workshop Job Scheduling Strategies for Parallel Processing (JSSPP '05)*, June 2005.
- [35] T. Xie and X. Qin, "A Security-Oriented Task Scheduler for Heterogeneous Distributed Systems," *Proc. 13th IEEE Int'l Conf. High Performance Computing (HiPC '06)*, Dec. 2006.



Xiao Qin received the BS and MS degrees in computer science from Huazhong University of Science and Technology, China, in 1996 and 1999, respectively, and the PhD degree in computer science from the University of Nebraska, Lincoln, in 2004. He is currently an assistant professor of computer science at Auburn University. Before joining Auburn University, he was with the New Mexico Institute of Mining and Technology. His research interests include parallel and distributed systems, real-time computing, storage systems, and performance evaluation. In 2007, he received a US National Science Foundation (NSF) Computing Processes and Artifacts (CPA) Award. He has served on the program committees of several conferences, including the IEEE International Conference on Cluster Computing (Cluster), IEEE International Performance, Computing, and Communications Conference (IPCCC), and International Conference Parallel Processing (ICPP). He is a member of the IEEE.



Tao Xie received the BS and MS degrees from the Hefei University of Technology, China, in 1991 and 2000, respectively, and the PhD degree in computer science from the New Mexico Institute of Mining and Technology in 2006. He is currently an assistant professor in the Department of Computer Science, San Diego State University, San Diego, California. His research interests are security-aware scheduling, high-performance computing, cluster and grid computing, parallel and distributed systems, real-time/embedded systems, and information security. He is a member of the IEEE.

► For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.