Abstract—We consider the problem of opportunistically scheduling low-priority tasks onto underutilized computation resources in the cloud left by high-priority tasks. To avoid conflicts with high-priority tasks, the scheduler must suspend the low-priority tasks (causing waiting), or move them to other underutilized servers (causing migration), if the high-priority tasks resume. The goal of opportunistic scheduling is to schedule the low-priority tasks onto intermittently available server resources while minimizing the combined cost of waiting and migration. Moreover, we aim to support multiple parallel low-priority tasks with synchronization constraints. Under the assumption that servers’ availability to low-priority tasks can be modeled as ON/OFF Markov chains, we have shown that the optimal solution requires solving a Markov Decision Process (MDP) that has exponential complexity, and efficient solutions are known only in the case of homogeneous behaving servers. In this paper, we propose an efficient heuristic scheduling policy by formulating the problem as restless Multi-Armed Bandits (MAB) under relaxed synchronization. We prove the indexability of the problem and provide closed-form formulas to compute the indices. Our evaluation using real data center traces shows that the performance result closely matches the prediction by the Markov chain model, and the proposed index policy achieves consistently good performance under various server dynamics compared with the existing policies.

I. INTRODUCTION

Resource utilization in the cloud typically exhibits a high level of dynamics, leaving much of the computation resources underutilized during off-peak periods. It is therefore essential to have a resource management system that can efficiently utilize the leftover cloud resources while ensuring minimal interference to the performance of existing workloads. A common solution to this problem is “valley filling”, i.e., filling the underutilized periods left by frontend tasks (e.g., web applications) with backend tasks (e.g., high performance computing applications). Protection to the frontend tasks can be implemented by suspending/evicting the backend tasks if the frontend workloads need to reclaim the resources. For example, in hypervisors using weights to resolve resource competition, valley filling can be implemented by giving zero (or some small) weight to the virtual machines (VMs) running backend tasks. As a result, the backend tasks cannot have guaranteed performance since they only have intermittent access to the resources, which hinders the adoption of valley filling in commercial clouds. It is therefore important to develop best effort scheduling mechanisms that can minimize the performance loss of backend tasks due to frontend interruptions.

Although resource sharing among jobs of different priorities has been well studied in the literature [1], most existing techniques are developed for a single machine environment where all jobs are scheduled jointly. In a distributed computing environment, the technique of cycle stealing [2] is developed to allow opportunistic scheduling of low-priority jobs. Designed for grid computing, cycle stealing emphasizes non-intrusiveness and low maintenance, and assumes the low-priority job will always wait at its host machine during busy periods.

The cloud environment brings both challenges and opportunities to this problem: On the one hand, resource-intensive cloud jobs, such as scientific computing, large-scale graph processing (Pregel [3]), and MapReduce applications [4], often contain tasks with synchronization requirements that spread over multiple servers (i.e., tasks for the same job must synchronize with each other and progress simultaneously). While such jobs are ideal candidates for backend tasks, it is difficult to schedule them opportunistically, as the available slots on individual servers depend on their own frontend activities and are rarely synchronous. On the other hand, the virtualized cloud environment allows us to manage tasks more actively. In particular, it is possible to migrate the VMs of interrupted backend tasks to other available servers instead of waiting indefinitely. Migration, however, comes at an operational cost due to the data and VM state transfer. Thus, judicious decisions must be made when using migration to improve the performance of backend tasks.

In our previous work [5], we studied this problem as an optimization of the combined cost of migration and waiting. We showed the hardness of computing the optimal scheduling policy and developed an efficient threshold policy in a special case. Specifically, we proved that the threshold policy is optimal when the server availability processes (i.e., binary time series indicating whether servers are available to backend tasks at given times) are i.i.d. across servers. In this paper, we focus on developing scheduling policies that have both low complexity and good performance in the general case. Our main contributions are as follows:

- We observe that the threshold policy is sensitive to heterogeneity in the server availability processes and quickly
deteriorates as these processes become more heterogeneous;
  • We cast the scheduling problem as a stochastic optimization
    problem known as restless Multi-Armed Bandits (MAB) [6]
    by relaxing the synchronization requirement. Based on this,
    we develop an index-based scheduling policy with only linear
    complexity and give explicit formulas to compute the indices;
  • We evaluate the index policy against existing scheduling
    policies including the threshold policy on two data sets obtained
    from real systems. We report that the performance on traces
    closely matches the theoretical prediction, and the proposed
    index policy provides significantly better robustness than the
    other policies throughout the tests, with particularly good
    performance in high heterogeneity cases.

The rest of the paper is organized as follows. Section II
formulates the problem. Section III presents the optimal
and the heuristic policies. Section IV evaluates the proposed
policies by trace-driven simulations. Section V reviews the
related work. Then Section VI concludes the paper.

II. PROBLEM FORMULATION

We now formalize our problem. In the sequel, we will only
focus on backend tasks and simply refer to them as tasks. We
will focus on the scheduling of a single set of parallel tasks.
Our solution can be applied repeatedly to schedule multiple sets
of tasks, although it is beyond our scope to study the order of
scheduling.

A. Workload Modeling

Consider a set of \( n \) parallel (backend) tasks that require
\( n \) servers (or VMs) to run, one per task. Assume periodic
scheduling, which implies a slotted time system where one
slot is the time between consecutive scheduling decisions. We
assume strong synchronization where the \( n \) tasks must run
simultaneously. This assumption approximates scenarios where
the time between synchronization points is much smaller than
a slot. It also guarantees the feasibility of our schedule in
scenarios with arbitrary synchronization requirements.

B. Resource Modeling

Given a pool of \( N \) (\( N \geq n \)) servers with time-varying
available resources (due to frontend workloads), we model their
availability for hosting the (backend) tasks by binary ON/OFF
processes \( \{a_t\}_{t=1}^{\infty} \), called the server availability processes. Here
\( a_t = (a_{h,t})_{h=1}^{N} \in \{0, 1\}^{N} \) is a vector of server states at slot
\( t \): \( a_{h,t} = 1 \) if server \( h \) is ON (i.e., available) at time \( t \),
and \( a_{h,t} = 0 \) otherwise. Without loss of generality, we assume each
server can only host one task\(^1\). We call a server “host” if it is
selected to host one of the tasks. Note that a host can be ON
or OFF (in which case the hosted task and all the other \( n - 1 \)
tasks must be suspended).

We study online scheduling where the exact server
availability in the future is unknown. Instead, we rely on
statistical models based on historical data. Our previous
study of an enterprise data center suggests that the server
availability processes can be modeled as ON/OFF Markov
chains [5]. Specifically, we found that the ON/OFF intervals
in the quantized utilization traces roughly follow the
Geometric distribution, as illustrated in Fig. 1, while the
ON/OFF intervals in Markov chains are exactly Geometrically
distributed. For simplicity, we assume the Markov chains to be
independent across servers, with known transition probabilities
\( \{p_{ij}\}_{i,j\in\{0,1\}} \), where \( p_{ij} \) is the probability that server \( h \) is in
state \( j \) given it was in state \( i \) in the previous slot. Markovian
models have been used as a compact description of time-
varying workloads for performance prediction in the cloud [7].
While the actual server availability processes may not be truly
Markovian, our evaluation on real traces shows that the model is
accurate in predicting scheduling performance (see Section IV).

\[ C \Delta \sum_{t=1}^{\infty} \beta^{t-1} c_t(m_t, w_t) \quad (1) \]

for a given discount factor \( \beta \in (0, 1) \). Our goal is to design
a scheduling policy that minimizes \( C \).

There are two main parameters in the above formulation: \( \gamma \)
and \( \beta \). The discount factor \( \beta \) is used to ensure finiteness of \( C \)
and give priorities to earlier slots; the undiscounted total cost
can be approximated by letting \( \beta \rightarrow 1 \). The per-migration cost

\(^1\)If a physical server can host multiple (backend) tasks, we can divide servers
into task-sized partitions and refer to these partitions as “servers”.

\(^2\)We assume the migration delay is much smaller than an average busy period
and is thus negligible compared with the waiting time.
\(\gamma\) is a system-dependent parameter that specifies how much a migration costs the cloud provider on the average relative to one slot of waiting, which depends on the task size, the network capability, the penalty for delayed completion, etc. Intuitively, \(\gamma\) controls the desirable tradeoff between migration and waiting. Rigorously, we have the following qualitative relationship: under the optimal scheduling policy, the total number of migrations \(m_1 \triangleq \sum_{t=1}^{\infty} \beta^{t-1} m_t\) monotonically decreases and the total waiting time \(w_1 \triangleq \sum_{t=1}^{\infty} \beta^{t-1} w_t\) increases with the increase of \(\gamma\) [5]. In the sequel, we will assume \(\gamma\) is given and focus on developing the cost-optimal policy.

### III. Scheduling Policies

The design of scheduling policies is complicated by the temporal dependency between slots. Consider, as an example, the *myopic policy* that only minimizes the immediate cost \(c_t(m_t, w_t)\). Simple calculation shows that it works as follows: migrate interrupted tasks on OFF servers (if any) to unselected ON servers (if any) if the number of such tasks is less than \(1/\gamma\); wait otherwise. Despite its low complexity, the myopic policy often fails to achieve optimality because it neglects the temporal dependency in scheduling decisions: our current scheduling decision (migrate/wait and where to migrate) will determine which servers act as hosts in the future (until changed by migration), and thus affect future performance. In previous work [5], we have shown that optimal consideration of such temporal dependency requires a computationally complex solution. In this section, we review the optimal solution and then propose an efficient heuristic solution.

#### A. Optimal Solution Based on Markov Decision Process

In [5], we showed that the scheduling problem can be cast as a Markov Decision Process (MDP) [8], where each scheduling decision minimizes a combination of the immediate cost in the current slot and the expected cost over a future time window.

Specifically, the MDP maintains a state \((x_t, a_t)\), where \(x_t\) denotes the current selection of hosts and \(a_t\) the states of all the servers as defined in Section II-B, both at slot \(t\). Here \(x_t = (x_{h,t})_{h=1}^N\) is a vector of indicators with values in \(\lambda \triangleq \{x \in \{0, 1\}^N : \sum_{h=1}^N x_h = n\}\), where \(x_{h,t} = 1\) indicates server \(h\) is a host at time \(t\) (totally \(n\) hosts). Let \(C(x_t, a_t)\) denote the minimum expected cost starting from state \((x_t, a_t)\). The *optimal schedule* is the solution to the following recursion:

\[
C(x_t, a_t) = \min_{x \in \lambda} \left[w(a_t, x) + \gamma m(x_t, x) + \beta E[C(x_{t+1}, a_{t+1})]\right], \tag{2}
\]

where \(w(a_t, x) \triangleq 1 - \beta \frac{1}{n} \sum_{h=1}^N x_{h,a_{h,t}}\) is the per-slot waiting cost, and \(m(x_t, x) \triangleq \frac{1}{2} \sum_{h=1}^N |x_{h,t} - x_h|\) the per-slot number of migrations, assuming the host selection is changed from \(x_t\) to \(x\). Under the new selection \(x\), the state transits by: \(x_{t+1} = x\), and \(a_{t+1}\) being the next state of the Markov chains describing server availability starting from state \(a_t\). Initial host selection is computed by \(x_1 = \arg\min_{x \in \lambda} C(x, a_1)\).

Although the recursion suggests a dynamic programming solution, the problem can be directly solved using standard MDP solvers such as linear programming. Nevertheless, the complexity is polynomial in the number of states, which is \(\binom{n}{N} 2^N\) in this case (\(\binom{n}{N}\) values for \(x_t\) and \(2^N\) values for \(a_t\)). That is, the complexity of the optimal policy is exponential in \(N\). This observation motivates us to explore efficient alternative solutions.

#### B. Heuristic Solution Based on Multi-Armed Bandits

To reduce the complexity of the policy, we leverage similarities between our problem and restless Multi-Armed Bandits (MAB) [6] to develop an efficient heuristic policy. MAB is a special MDP where the controlled system consists of \(N\) subsystems, called *bandits*, which evolve independently, and the controller takes active actions for \(n\) \((n \leq N)\) out of the \(N\) bandits and passive actions for the rest. In classic MAB, passive bandits do not change states, whereas in restless MAB, both active and passive bandits may change states, possibly by different rules. While MAB always has an efficient solution, the same holds only for certain restless MABs that satisfy an indexability condition; in both cases the solution is an index policy that computes a scalar value for each bandit depending on its current state, called the index, and takes active actions only on the bandits with the top \(n\) indices.

Our problem shares a similar structure as MAB, where each server forms a bandit; this bandit is restless as servers can change availability states even if not used by the scheduler (due to frontend activities)\(^3\). In the language of MAB, we rewrite (2) into an equivalent reward maximization problem

\[
R(x_t, a_t) = \max_{x \in \lambda} \left[R(x_t, a_t, x) + \beta E[R(x_{t+1}, a_{t+1})]\right], \tag{3}
\]

where \(R(x_t, a_t) \triangleq 1/(1 - \beta) - C(x_t, a_t)\). The per-slot reward function is given by:

\[
R(x_t, a_t, x) \triangleq \frac{1}{n} \sum_{h=1}^N x_{h,a_{h,t}} - \gamma \sum_{h=1}^N |x_{h,t} - x_h|, \tag{4}
\]

While MAB requires the reward to be additive across bandits, the reward in (4) is clearly not additive over \(h = 1, \ldots, N\) for any \(n > 1\). We thus approximate it by removing the \(\lfloor \cdot \rfloor\) operator, which relaxes \(R(x_t, a_t, x)\) into another reward \(\tilde{R}(x_t, a_t, x)\) that can be decomposed into single-bandit rewards. Specifically, we can write it as \(\tilde{R}(x_t, a_t, x) = \sum_{h=1}^N R^h(x_{h,t}, a_{h,t})\), where

\[
R^h(x, a) \triangleq \frac{a}{n} - \gamma (1 - x), \quad R^h_0(x, a) \triangleq 0 \tag{5}
\]

are the active and passive per-slot reward for the single bandit representing server \(h\) (here migration cost is only counted at the new host). Operationally, this relaxation means that instead of requiring all the \(n\) hosts to be ON simultaneously, the tasks can now be processed independently, each completing \(1/n\) unit workload per slot if its host is ON.

\(^3\)In contrast, even though the formulation in [9] makes the problem of stochastic scheduling with switching cost a special restless MAB, the original problem has rest and is inherently different from ours.
The above relaxation reduces the problem to a standard restless MAB, where Whittle’s index policy [6] gives an efficient and near-optimal solution. The challenge is that it is generally hard to test indexability and compute the index. Fortunately, our problem turns out to satisfy a stronger condition called generalized conservation laws (GCL) [10], which not only guarantees Whittle’s indexability but also allows us to compute the index explicitly.

Specifically, for a single bandit with state space $N$, define $T^S_i$ as the total expected discounted time the bandit lies in $S \subseteq N$ with initial state $i \in N$, if we take active action whenever its state falls into $S$. For a given $S$, the values of $T^S_i$ can be computed by solving the system of equations:

$$T^S_i = 1 + \beta \sum_{j \in N} p_{ij} T^S_j, \quad i \in S,$$  \hspace{1cm} (6)

$$T^S_i = \beta \sum_{j \in S^c} p_{ij} T^S_j, \quad i \in S^c,$$  \hspace{1cm} (7)

where $p_{ij}, p_{0j}$ are the state transition probabilities under active/passive action. Using $T^S_i$, define quantities $A^h_i$ by

$$A^h_i = 1 + \beta \sum_{j \in N} (p_{ij} - p_{0j}) T^S_j.$$  \hspace{1cm} (8)

Using $A^h_i$ and active rewards $R^h_i = \Delta (R_i)_{i \in N}$, [10] gives an $O(|N|^{3})$-complexity adaptive greedy algorithm AG, which will compute indices $\alpha = (\alpha_i)_{i \in N}$ for each state, and determine if $R^h_i$ is admissible. The GCL theory states the following.

**Theorem III.1** (GCL-indexability [10]). 1) A restless bandit is GCL-indexable if: (i) $A^h_i > 0$ for all $i \in N$, $S \subseteq N$, and (ii) $R^h_i$ is admissible by AG; 2) A GCL-indexable bandit is always indexable, and its Whittle’s indices are $\alpha$ computed by AG.

Applying this theorem to our single bandit (5), we obtain the following closed-form solution to Whittle’s indices.

**Corollary III.2.** The single bandit (5) is GCL-indexable, hence indexable. Its Whittle’s indices are given by: $\alpha^h_{1,1} = 1/n$, $\alpha^h_{0,0} = -\gamma(1-\beta)$, and

$$\alpha^h_{1,0} = \frac{\beta p_{01}^h}{n(1 - \beta p_{11}^h + \beta p_{01}^h)},$$  \hspace{1cm} (9)

$$\alpha^h_{0,1} = \frac{1 - \beta p_{01}^h}{n(1 - \beta p_{11}^h + \beta p_{01}^h)} - \gamma(1 - \beta)$$  \hspace{1cm} (10)

if $\gamma \geq \frac{1}{n(1 - \beta p_{11}^h + \beta p_{01}^h)}$, and

$$\alpha^h_{0,1} = \frac{1}{n} - \gamma(1 - \beta p_{11}^h), \quad \alpha^h_{1,0} = \gamma p_{01}^h$$  \hspace{1cm} (11)

otherwise.

**Proof:** See the proof of Corollary 2.6 in [11].

This result gives us an index-based scheduling policy as follows (assuming $x_1 = 1$ to ignore the migration cost for the initial host assignment): at each time $t$, the policy

1) uses the above formula to evaluate the Whittle’s index for each server based on its current state $(x_{h,t}, a_{h,t})$, and then

2) schedules the tasks onto the $n$ servers with the maximum indices, migrating tasks if necessary.

Note that although in deriving the policy, we reward each host independently, in applying the policy, we still require all the $n$ hosts to be simultaneously ON to run the tasks.

A byproduct of this solution is an upper bound on the optimal reward $R^*$ given by the MDP (2) (or a lower bound on the optimal cost $C^*$). It is easy to see that the optimal relaxed reward $\bar{R}$ upper bounds $R^*$, and Whittle’s policy is known to be near optimal in approximating $\bar{R}$ (for large $N$). Therefore, if $R^W$ denotes the reward of Whittle’s policy under the original reward function and $R^W$ the reward under the relaxed reward function, one can expect that $R^W \leq R^* \leq R^W$ (or $C^W \leq C^* \leq C^W$ in terms of cost). This is particularly useful for evaluating the suboptimality of heuristic policies at large $N$, when it is impractical to solve the MDP to compute $R^*$ (or $C^*$) exactly.  

**Remark:** The closed-form solution reveals insights into the scheduler’s behavior. For each bandit, the ordering of indices is $\alpha^h_{1,1} \geq \alpha^h_{0,1} \geq \alpha^h_{0,0}$ if $\gamma > 1/(n(1 - \beta p_{11}^h + \beta p_{01}^h))$, and $\alpha^h_{1,1} \geq \alpha^h_{0,1} \geq \alpha^h_{0,0}$ otherwise. Our first observation is that $\alpha^h_{1,1}$ is always the largest and $\alpha^h_{0,0}$ always the smallest among all indices of all the bandits, regardless of the server. This implies that the index policy will never migrate away from an ON server, or migrate to an OFF server.

In choosing between the other two options: wait at current host ($(x_{h,t}, a_{h,t}) = (1, 0)$) or migrate to another ON server ($(x_{h,t}, a_{h,t}) = (0, 1)$), the policy will depend on the server dynamics $p_{ij}^h$. For i.i.d. servers ($p_{ij}^h = p_{ij}, \forall h$), the policy is reduced to one of the extreme cases with a threshold on $\gamma$: if $\gamma < 1/[n(1 - \beta p_{11}^h + \beta p_{01}^h)]$, then $\alpha^h_{1,1} > \alpha^h_{0,0}$, and the policy will always migrate tasks out of unavailable hosts; if $\gamma \geq 1/[n(1 - \beta p_{11}^h + \beta p_{01}^h)]$, then $\alpha^h_{1,0} \geq \alpha^h_{0,1}$, and the policy will always wait when some of the hosts become unavailable.

**C. Closed-form Optimal Solution in a Special Case**

In the special case that all servers behave homogeneously, i.e., the availability processes are i.i.d. across servers, and there are always at least $n$ ON servers, we have developed a closed-form solution for optimal scheduling [5]. The solution is a threshold policy that will wait if the number of uninterrupted tasks is less than a threshold $\tau \in \mathbb{N}$ (i.e., the number of interrupted tasks $\tau - n - \tau$, and migrate interrupted tasks to unselected ON servers otherwise. This is very simple policy with near constant complexity (to count the number of interrupted tasks; assume $n \ll N$), and is yet provably optimal. Compared with the myopic policy introduced at the beginning of Section III, we see that both policies impose a threshold on the number of interrupted tasks, but the thresholds are different ($1/\gamma$ for the myopic policy, $n - \tau$ for the threshold policy). The calculation of threshold for the threshold policy involves sophisticated optimization that depends on not only $\gamma$ but also server dynamics; see Appendix.

**IV. Performance Evaluation**

Our evaluation mainly targets at two questions: (1) Since we have derived the policies based on the Markov chain
model of server availability, how do these policies perform on real data and how do the performances fit those predicted by the model? (2) How does the heuristic policy based on Whittle’s index compare with existing policies? In particular, can it provide satisfactory performance in cases when the threshold policy lacks performance guarantee (i.e., for servers with heterogeneous availability patterns)?

To this end, we simulate the proposed policies on availability traces derived from real measurements in two different environments. The first environment (referred to as “PERFORM”) is a pool of servers hosting business users, where we collect CPU utilization traces at 15-minute intervals and convert them into availability traces using a threshold (assumed to be available if utilization \( \leq 15\% \)). The second environment (referred to as “SETI”) is a pool of home computers used by private users [12], where we use an existing set of traces indicating the availability of each computer for desktop grid applications. The original SETI traces are continuous in time, and we convert them into slotted traces using 15-minute slot. All the simulations last for at least 1000 slots (only traces \( \geq 1000 \) slots are considered). Assume \( \beta = 0.999 \).

The two data sets exhibit quite different availability behaviors. As shown in Fig. 2, the per-server availability of SETI traces is near uniformly distributed, whereas that of PERFORM traces is concentrated around 0% and 100%, with only half of the traces showing availability in between. The distributions of ON/OFF interval lengths are also different.

![Fig. 2. Distribution of (a) fraction of availability and (b) average ON/OFF interval length across servers.](image)

A. Evaluation on PERFORM Traces

We preprocess the traces to filter out degenerate ones (i.e., almost always ON/OFF) to focus on servers with sufficiently dynamic availability, as shown in Fig. 3. To test homogeneous scenarios, we select three representative sets of servers with low, medium, and high availability, respectively, as illustrated in Fig. 3. We keep the servers in each set relatively homogeneous in both transition probabilities and availability/ON interval length. To test heterogeneous scenarios, we fix \( N \) and randomly select \( N \) servers from a larger pool of heterogeneous servers.

1) Simulations at Small Scale: We first simulate the policies at small scale. Due to the high complexity of MDP, the optimal policy can only be computed for small \( N \) and \( n \) (here \( n = 2 \), \( N = 6 \)). Thus, small-scale simulations will allow us to compare the more efficient alternative policies with the optimal policy to characterize their suboptimality.

2) Simulations at Increased Scale: We then increase the scale to \( n = 10 \) and \( N = 100 \) for a more realistic test. We repeat the above simulations for all the policies except the optimal policy, which is too complex to compute. From the average performance in Fig. 8, the index policy mildly improves the performance of the existing policies. As we examine specific sets of servers, however, we observe that the index policy gives consistent performance overall all the server sets, whereas the threshold policy and the myopic policy can perform substan-
Fig. 4. Total cost vs. $\gamma$: PERFORM, small-scale, near-homogeneous servers, low availability, averaged over all the (7) subsets of 6 servers.

Fig. 5. Total cost vs. $\gamma$: medium availability (rest as in Fig. 4).

Fig. 6. Total cost vs. $\gamma$: high availability (rest as in Fig. 4).

Fig. 7. Total cost vs. $\gamma$: PERFORM, small-scale, heterogeneous servers, averaged over 10 sets of randomly selected servers (6 servers per set).

Fig. 8. Total cost vs. $\gamma$: PERFORM, increased scale, averaged over 10 sets of randomly selected servers (100 servers per set).

Fig. 9. Total cost vs. $\gamma$: PERFORM, increased scale, 100 highly heterogeneous servers.

Fig. 10. Selected servers: 247 heterogeneous servers ($\circ$); near-homogeneous servers with low (○), medium (●), and high (△) availability; 7 servers each.

A) Simulations at Small Scale: We first repeat the small-scale simulations on SETI traces; see Fig. 11–14 ($n = 2, N = 6$). We still see a transition from active migration (positive slope) to no migration (zero slope) with the increase of $\gamma$. The main transition point, however, is at a much larger $\gamma$ than in PERFORM tests. This is because SETI traces have much longer ON/OFF intervals than PERFORM traces, making it more worthwhile to migrate interrupted tasks. The threshold policy has similar performance as before, near optimal in approximately homogeneous cases (Fig. 11) and suboptimal in heterogeneous cases (Fig. 14). It now shows a bigger advantage over the myopic policy because the myopic policy ignores temporal correlation while SETI traces exhibit strong temporal correlation. Note that SETI traces are more scattered at higher availability levels (Fig. 10 (b)), making it harder to select homogeneous servers with medium–high availability; this explains the gap between the threshold policy and the optimal policy in Fig. 12–13 (as the selected servers are more heterogeneous than those in Fig. 11). The index policy again gives consistently good performance, especially in the heterogeneous cases.
In summary, while the threshold policy is near optimal for servers with homogeneous availability behaviors, the index policy provides a robust, equally efficient alternative in heterogeneous cases when the threshold policy degrades.

**V. RELATED WORK**

Opportunistic scheduling has been used to share resources between jobs of different priorities while being non-intrusive to high-priority jobs. The emphasis is thus on optimizing the performance of low-priority jobs for given available resources. Depending on the knowledge of resource availability, existing work can be divided into two categories: (i) offline opportunistic scheduling, where the scheduler of low-priority jobs has complete knowledge of the available resources throughout time (e.g., as in systems with advance reservation) [13], and (ii) online opportunistic scheduling, as in this paper, where the knowledge is limited to past observations and long-term statistics. In online scheduling, approaches to improving the performance include building sophisticated resource models [12] and managing low-priority jobs more actively (e.g., using live migration) [5].

VM migration has been extensively studied in the cloud context, e.g., [14], [15] and references therein, where the main focus has been to understand the performance of migration [15] or to improve the performance [14]. In this paper, we focus on a different aspect of when to evoke migration compared with the alternative of waiting.

Waiting as a control option has been studied in a different context [16], where cloud users have to trade off job waiting time with the monetary cost of resources. In contrast, our problem is from the cloud provider’s perspective, where the tradeoff of interest is different.

Technically, our approach is most related to a stochastic model of Multi-Armed Bandits (MAB) and has received...
much attention due to its broad applicability. Specific results include index-based policies to handle switching costs/delays and precedence relations; see [9] and references therein. In particular, [9] reformulates the problem into a restless MAB without switching cost, based on which [18] proposes a two-staged algorithm to compute the indices efficiently. Recent work [19] extends the problem to include stochastic arrivals and queueing dynamics. Our problem is a dual to this problem in that we schedule tasks among multiple servers, with three crucial differences: (a) our scheduler does not have to fully utilize all the servers; (b) while unattended jobs remain the same, unutilized servers can change states due to frontend activities; (c) in contrast to independent jobs in previous work, we consider parallel tasks that must run simultaneously.

Another related problem is work stealing [20], where idle resources steal work from busy ones. It differs from our problem in that it assumes dedicated (i.e., non-shared) resources, and there is no cost of stealing.

VI. Conclusion

We have studied the problem of scheduling parallel (backend) tasks onto opportunistically available server resources with focus on the tradeoff between migration and waiting. After reviewing the limitation of existing solutions, we propose a heuristic scheduling policy based on Whittle’s index that greatly reduces the complexity of the optimal policy while achieving consistently good performance under a variety of server dynamics. Although derived under simplifying assumptions (Markovian availability processes, relaxed synchronization), the index policy is shown to be effective and robust in extensive simulations based on traces from two different environments.

REFERENCES


VII. Appendix

We include the calculation of threshold for the threshold policy (see Section III-C) for completeness. Let \( p_{ij} \), \((i, j \in \{0, 1\})\) denote the transition probability of server states (recall the probabilities are identical across servers). Define \( P = (P_{ij})_{i,j=0}^n \) as the transition matrix of host states, where \( P_{ij} \) is the probability that \( j \) out of the \( n \) hosts are ON in the current slot given \( i \) of them were ON in the previous slot. We have

\[
P_{ij} = \min_{k=0}^{\min(i,j)} \binom{i}{k} p_{11}^k (1 - p_{11})^{i-k} \binom{n-i}{k} \beta_{j-k} (1 - \beta_{j-k})^{n-i-j+k}.
\]

Define \( Q = (Q_{ij})_{i,j=0}^n \) as the migration count matrix, where \( Q_{ij} = |i - j| \) is the minimum number of migrations to change the number of ON hosts from \( i \) to \( j \) (by changing the hosts). Let \( \lambda = (\lambda_i)_{i=0}^n \) denote the initial distribution of host states, where \( \lambda_i \) is the probability for \( i \) out of the \( n \) initial hosts to be ON. Denote an arbitrary scheduling policy by \( M = (M_{ij})_{i,j=0}^n \), where \( M_{ij} \) is the probability that the policy will change the number of ON hosts from \( i \) to \( j \) (within the same slot) by migration (\( \sum_{j=0}^n M_{ij} = 1 \) for all \( i \)). For example, a threshold policy with threshold \( \tau \) is represented as \( M^\tau \) with \( M_{ii}^\tau = 1 \) for \( i = 0, \ldots, \tau - 1 \) and \( M_{ii}^\tau = 1 \) for \( i = \tau, \ldots, n \).

The algorithm for computing threshold \( \tau \) works as follows:

1) for \( i = 0, \ldots, n \), compute \( \pi_i^T = \lambda^T M^i (1 - \beta M^i)^{-1} \), \( P_i = (\pi_i)_n \), and \( M_i = (\lambda^T + \beta \pi_i^T) P_i (M_i^T) \); 

2) for \( i = 1, \ldots, n \), compute \( \gamma_i = \frac{P_i - P_{i-1}}{(M_i - M_{i-1})}; \) \( \gamma_0 = 0, \gamma_{n+1} \triangleq \infty; \)

3) the threshold is \( \tau = i \) if the per-migration cost satisfies \( \gamma_i \in [\gamma_i, \gamma_{i+1}) \) (\( \tau \) is either \( i \) or \( i - 1 \) if \( \gamma = \gamma_i \)).

The operational meanings of \( P_i \) and \( M_i \) are the total (discounted) running time and the total number of migrations under the threshold policy with \( \tau = i \). See [5] for the derivation.

---

4A stealing delay has been considered, but it is not associated with job interruption or any other cost.