iService: A Cloud-based Scheduling Service for Efficient Usage of IoT Resources

Abirami Sankara Narayanan, Yang Peng and Brent Lagesse
University of Washington Bothell, Bothell, WA 98011
{abirami2, yangpeng, lagesse}@uw.edu

***Abstract*—With the proliferation of the Internet of Things (IoT), numerous IoT devices have been deployed in various living spaces such as university campuses, community centers, offices and homes. Users of IoT services are more and more willing to explore and directly interact with interested devices over mobile or web applications. However, there is a lack of online service that can efficiently schedule requests issued by various IoT users to distributed IoT devices. Efficient, cost-effective scheduling is critical to the user experience in a smart living space. In this paper, we propose iService, a cloud-based scheduling service for efficient usage of IoT resources given IoT-specific factors, such as device location, sensing capability, and energy cost. In our solution, the essential scheduling problem has been formulated as a mixed integer nonlinear programming (MINLP) problem. To efficiently address this problem in an online fashion, iService has been implemented using multiple AWS technologies. Via extensive experiments, the effectiveness of iService has been demonstrated under different request patterns and system parameters. From these experiments, we conclude that iService can improve the quality of service to the user and thus their quality of life within a pervasive living space.**

I. Introduction

With the rapid growth of Internet of Things (IoT), numerous IoT devices are expanding into various living spaces, such as smart homes, smart offices, smart campuses, and smart cities. Because of the convenience and intelligence provided by IoT services, interacting with these IoT devices via mobile or web applications has become a routine activity in our daily lives. In many places such as university campuses and company offices, many heterogeneous IoT devices are usually deployed to provide a portfolio of services, which are typically used in a shared manner by a large number of users. For example, an IoT printer may provide a "remote print" service for a group of students, an autonomous robot may provide a "delivery" service to targeted offices, and a surveillance camera may provide a "safety check" service. To use these shared IoT services, users need to submit requests to avoid time conflicts or resource limitations. However, at peak times or for popular services, it is difficult to obtain consistent quality of services due to user competition and inefficient usage of IoT resources. This problem may become more severe when the number of users increases or when the dependency between services becomes complicated. Therefore, efficient scheduling tasks shall be performed to make the best usage of IoT resources given diverse user requirements.

It has been shown by numerous studies across several different types of systems that excessive delay is one of the greatest sources of user frustration [1], [2]. Additionally, it has been shown that such delay not only decreases the users' quality of experience, but also reduces the effectiveness of the system (e.g. slow learning in educational systems or less participation in community-focused systems) [3]. Furthermore, frustrating delays have also been shown to reduce user's perceived quality of a system [4]. To truly provide a pervasive living space that enhances the participants' experience, we must be able to schedule tasks with minimal delay.

Essentially, such a scheduling problem can be recognized as an unrelated parallel machine (UPM) scheduling problem. In the classic UPM scheduling problem, machines are considered unrelated when the processing time of jobs purely depends on the machine to which they are assigned, and there is no relationship between each machine's processing speed. Similarly, IoT devices can be interpreted as unrelated machines whereas IoT requests from users are the jobs to be scheduled on machines. Research about the UPM scheduling problem has been widely discussed in the literature, and most of them targeted on the minimization of total completion time [5], [6], total tardiness [7] and makespan [8]. Although these traditional scheduling methods may be employed to address the need of IoT service scheduling, they are lack of the consideration of unique scheduling factors such as device location, sensing capability, and energy cost for IoT.

To fill this gap and make efficient usage of IoT resources, we propose a cloud-based scheduling service in this paper, called iService. When designing the core scheduling scheme of iService, we explicitly consider unique IoT factors such as device heterogeneity, energy cost, device location, number of concurrent users allowed by a service, and short execution time of an IoT service. Such an IoT-unique scheduling problem has been formalized as a mixed integer nonlinear programming problem (MINLP). To satisfy the online scheduling needs, iService has been implemented using multiple AWS technologies and hosted in cloud as a scalable web service. In particular, an MINLP solver has been employed by iService to handle varying service requests and make the most efficient schedule of them to utilize distributed IoT resources. Figure 1 illustrates iService's high-level design. To the best of our knowledge, this is the first work on studying IoT-specific service scheduling problem and implementing the scheduling framework as an online service.

The rest of this paper is organized as follows. Section II describes the system model and formally present the studied problem. The design and implementation details of iService are discussed in Section III, and the performance evaluation results obtained from various experiments are shown in Section IV. Related work on job scheduling algorithms are reviewed in Section V. Section VI concludes the paper and discusses future research avenues.

978-1-5386-9151-9/19/$31.00 ©2019 IEEE

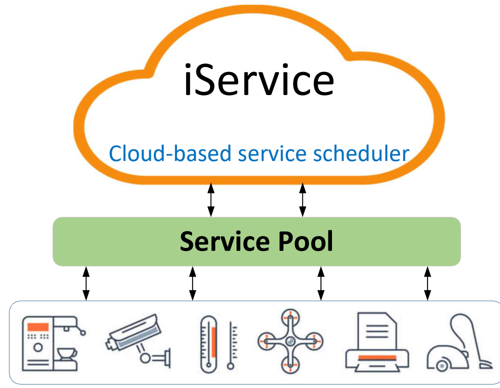

Fig. 1. High-level design of iService

II. Analysis

In this section, we describe the system model and formally present the problem of IoT service scheduling.

A. System Model

The considered system consists of four key components, device, service, request, and scheduler.

- An IoT device d_i is equipped with one or more sensors, and it is deployed at location (x_i, y_i). Given its available energy level e_i at the scheduling time, d_i may support a set of IoT services. D is the set of devices that can be accessed by a scheduler.
- An IoT service s_i is of various types of operations supported by an IoT device. When a service s_i is running on device d_j, it may be completed without interruption in $\lambda_{i,j}$ seconds at an energy consumption of $\delta_{i,j}$ joules. Additionally, service s_i can only support up to $u_{i,j}$ number of concurrent users on device d_j. Different devices many support the same type of service, but the corresponding processing time, energy consumption and number of allowed users may vary. S is the set of services supported in the system.
- A request r_i can ask to use a group of services available on some device in an interested area centered at (x_i, y_i) with a radius of ϕ_i. All the services in a request group must be of different types and sequentially executed on the same device, and multiple requests may be executed on the same device simultaneously. R is the set of requests a scheduler needs to schedule.
- A scheduler keeps running in the system. At time t, it schedules a set of requests R to execute on any device in D. A request r_i arrives (having been accepted by the scheduler) at time a_i with an expected completion time of c_i, which may be different from the actual completion time. The processing time of request r_i is denoted as p_i, which depends on r_i's preceding requests scheduled to be executed on the same device.

B. Problem definition

Given the models described above, we can formally define the scheduling problem as follows.

- Objective:
  $\min\ \{M\}$, where M can be defined as
  - either the maximum makespan of all requests: $M = \max\ \{t + p_i - a_i\}$, $r_i \in R$,
  - or the maximum tardiness of all requests: $M = \max\ \{t + p_i - c_i\}$, $r_i \in R$.
- Given:
  - A: arrival time vector, where a_i is request r_i's arrival time.
  - C: expected-completion time vector, where c_i is request r_i's expected completion time.
  - SD: service-device matrix, where $sd_{i,j}$ indicates service s_i is available on device d_j.
  - SR: service-request matrix, where $sr_{i,j}$ indicates service s_i needs to be used by request r_j.
  - Δ: energy consumption matrix, where $\delta_{i,j}$ indicates the total energy consumption to complete all of request r_i's services on device d_j.
  - Λ: processing time matrix, where $\lambda_{i,j}$ indicates the total processing time to complete all of request r_i's services on device d_j.
  - E: device energy vector, where e_i is device d_i's energy level at the scheduling time.
  - L: device location vector, where l_i contains device d_i's location (x_i, y_i).
  - RL: request location vector, where rl_i contains request r_i's requested center of location (x_i, y_i).
  - Φ: radius vector, where ϕ_i is the radius allowed by request r_i.
  - U: user matrix, where $u_{i,j}$ is the number of concurrent users allowed by service s_i on device d_j.
- Output:
  - W: decision matrix, where $w_{i,j} = 1$ if request r_i is scheduled to execute on device d_j; 0 otherwise.
  - Π: precedence matrix, where $\pi_{i,j} = 1$ if request r_i shall be executed before r_j. In particular, $\pi_{i,i} = 1$ if request r_i is the first one to be executed.
- Subject to:
  - Processing-time constraint, $\forall r_i \in R$
    - $p_i = \sum_{r_h \in R, h \neq i} \pi_{h,i} \cdot p_h + \sum_{d_j \in D} w_{i,j} \cdot \lambda_{i,j}$
  - Energy constraint, $\forall r_i \in R$
    - $\sum_{d_j \in D} w_{i,j} \cdot \delta_{i,j} \leqslant e_j$
  - Location constraint, $\forall r_i \in R$
    - $\sum_{d_j \in D} w_{i,j} \cdot \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \leqslant \phi_i$
  - User constraint, $\forall s_i \in S$
    - $\sum_{r_k \in R, r_h \in R} sr_{i,k} \cdot w_{k,j} \cdot \pi_{h,k} \leqslant u_{i,j}$
  - Single-device constraint, $\forall r_i \in R$
    - $\sum_{d_j \in D} w_{i,j} = 1$
  - Single-precedence constraint, $\forall r_i \in R$
    - $\sum_{r_k \in R} \pi_{k,i} = 1$

In this formalized problem, Δ and Λ can be computed using matrices of SD and SR along with each service's processing and energy consumption information. The processing-time

constraint specifies request r_i's processing time is dependent on the processing time of all requests scheduled before it (i.e., limited by $\pi_{h,i}$) on the same device. The user constraint specifies the total number of requests using the same service s_i (i.e., indicated by $sr_{i,k}$) on the same device d_j must not be over the limit $u_{i,j}$.

Obviously, many of the input and output variables in the definition are 0-1 integer numbers. Moreover, the constraints are nonlinear in nature with the goal of minimizing the objective function. Therefore, this scheduling problem can be classified as a MINLP problem, which is NP-hard. In order to solve this problem, we employ an MINLP solver when implementing iService, the details of which will be presented in Section III-B.

III. Design and Implementation

In this section, we discuss the design and implementation details of iService.

A. Design overview

Our design goal is to develop an auto-scale cloud-based service which can handle time-varying volumes of IoT requests from different users. To achieve this goal, multiple AWS technologies are utilized for their intrinsic support of manageability and scalability. Figure 2 shows the key components in the design of iService, and the overall workflow is as follows.

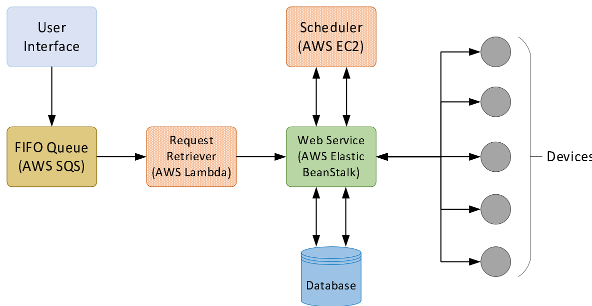

Fig. 2. Key components of iService

- When an IoT request is submitted, the front-end UI validates the request and sends it in JSON format to a First-In-First-Out (FIFO) queue implemented using Amazon Simple Queue Service (AWS SQS). This way, user requests can be buffered and retrieved in their exact arrival order without a loss, even at a high volume.
- Later, an AWS Lambda function (invoked either periodically or on-demand per user's specification) retrieves the request from SQS and post it to a RESTful web application, which is deployed on AWS Elastic Beanstalk. In our implementation, AWS Lambda is utilized to automatically trigger other AWS services, which saves effort on provisioning and managing servers. Additionally, we use AWS Elastic Beanstalk to host our RESTful web application, because AWS Elastic Beanstalk can automatically handle web service jobs such as deployment, capacity provisioning, and load balancing. The RESTful web application's main responsibilities are to validate and classify user requests, construct data files for the scheduler, invoke scheduler, and retrieve and execute scheduling decisions.
- Finally, a remote connection is established between the AWS Elastic Beanstalk and the AWS EC2 instance where the scheduler is installed to process requests. The scheduler running on EC2 is responsible for allocation of IoT requests to devices and exchanges of scheduling information with the web application. Based on the output from the scheduler, iService assigns individual IoT requests to selected devices, and logs the execution history into a MySQL database.

B. Scheduler

To solve the formalized scheduling problem, we install AMPL [9] and Bonmin [10] solver on an Amazon EC2 Linux instance, which effectively serves as a scheduler.

In this scheduler, Bonmin is the core solver, which features several algorithms such as the "branch and bound" algorithm, "branch and cut" algorithm, etc. As our problem is identified as a MINLP problem, the "branch and bound" algorithm provided by Bonmin is an efficient solution for MINLP problems and thus is configured as the default algorithm of the scheduler.

AMPL, on the other hand, enables high-level algebraic representation of an optimization problem. With the help of AMPL, iService can effectively decouple dynamic values of input variables (as presented in Section II-B) and static model files (which do not change given different input parameters). Figure 3 illustrates the scheduler's internal modules along with the inputs and output variables.

In order to efficiently use AMPL, iServie creates model and data files at different time and uses them as follows to complete a scheduling task.

- *Model file.* This file contains the core logic for solving the problem. We define the objective, input parameters, output variables, and constraints in it. When implementing iService, four different model files have been prepared and stored on the EC2 instance. In addition to the files for the objectives of "minimizing the maximum makespan" and "minimizing the maximum tardiness," the model files for the objectives of "minimizing the sum of makespan" and "minimizing the sum of tardiness" have also been defined, all of which can be used according to the web application's configuration.
- *Data file.* This file contains the input data needed by the model file. When the web application triggers the scheduler to execute, this data file is generated at run-time using the information of requests to be scheduled. For example, the service-request matrix, the device energy vector, and other dynamically changed information with each batch of requests can be captured in this data file, and feed to the static model file that has already been placed on the scheduler's EC2 instance.

By using separate model and data files to describe the scheduling procedure, the web services and scheduler program can operate independently and also adapt to changes of use requests.

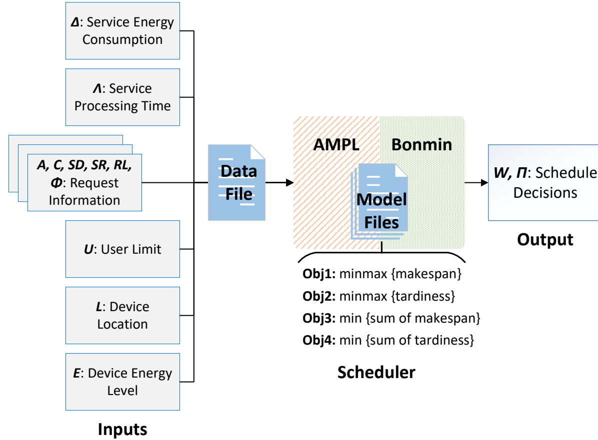

Fig. 3. Input and output of scheduler

C. States in request processing

To track the status of each request, iService periodically updates the states of all incomplete requests by checking their arrival time, scheduled start time, and scheduled completion time. The transitions between states are depicted in Figure 4.

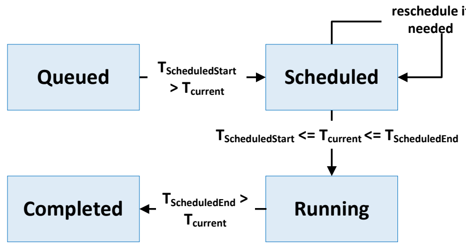

Fig. 4. States in processing of requests

- *Queued.* When a user provides the request details and submits the request for processing, the request is pushed to queue and enters the "queued" state. All requests in queued state will be considered for scheduling when their information is posted to the web application.
- *Scheduled.* In this state, the request has been scheduled to run on a particular device but the device is busy processing other requests that are scheduled prior to this request. This type of requests can also be considered for rescheduling, when the scheduler checks if there is a possibility to serve this request by some other device so as to further optimize the scheduling objective.
- *Running.* In this state, the request is being processed by some device and it will not be interrupted or considered for further scheduling process.
- *Completed.* When the device completes processing a request, the request is said to be in completed state.

When a request's state is updated, the corresponding user count, as well as available energy, are updated in database.

IV. Performance Evaluations

Extensive experiments have been conducted to evaluate the performance of iService with varying request pattern and device availability.

- Request pattern. The request pattern depends on the services requested by users. If all the requests to be scheduled demand the same set of services, the request pattern is considered homogeneous. Whereas if all the requests demand different set of services, the pattern is considered heterogeneous.
- Device availability. This is defined as the ratio of devices available to serve IoT requests at any point of time. When a device's energy is too low, it is considered unavailable.

Each of these two parameters, along with the number of requests to be scheduled, can affect the search space of the employed solver, and thus significantly affect the computing time for the scheduler to find the optimal schedule.

A. Evaluation Results

In the first set of experiments, we evaluated the computing time for the scheduler to find the optimal schedule (feasible solution) under homogeneous and heterogeneous request patterns. From Figures 5 and 6, we can find that it takes more time to find the optimal schedule when the number of requests increases. Additionally, the computing time is consistently longer given heterogeneous request patterns. This is because requests of heterogeneous pattern use a more diverse set of services, which creates a larger search space. Therefore, it takes longer time to find the best solution. From these results, we can get the insight that it is critical to limit the size of request group for every scheduling activity, such that a low computing time will be needed and thus users only experience short service delay.

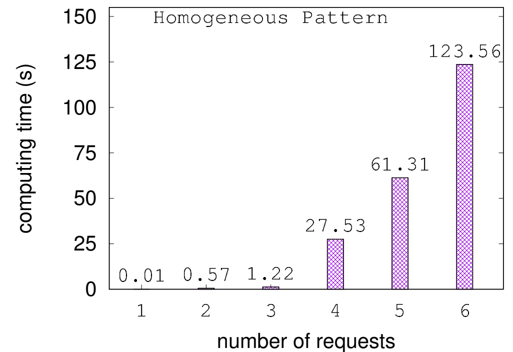

Fig. 5. Computing time for homogeneous request pattern

In the second set of experiments, we varied the device availability to evaluate its effect on the time to find the best schedule. For these experiments, the number of requests is fixed to three and each request asks for one service. Initially, the experiment was conducted by setting 25% of devices online. This resulted in finding a feasible solution in a much short time. When the availability increases, it can be seen from Figure 7 that the computing time increases for both

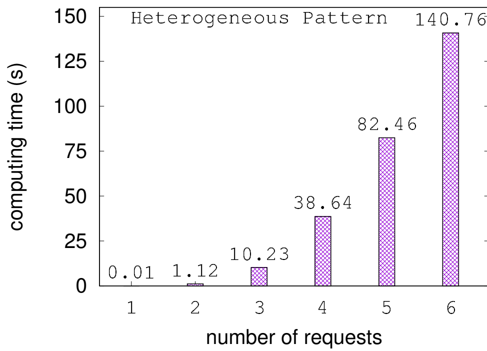

Fig. 6. Computing time for heterogeneous request pattern

homogeneous and heterogeneous request patterns. This is due to the fact that the scheduling algorithm must explore a larger search space when more devices are online. From these experiments, we can learn that certain filtering mechanisms based on device similarity, energy level or shall be performed before handing over requests to scheduler. This way, the service delay can be practically reduced without affecting the service quality provided by needed devices.

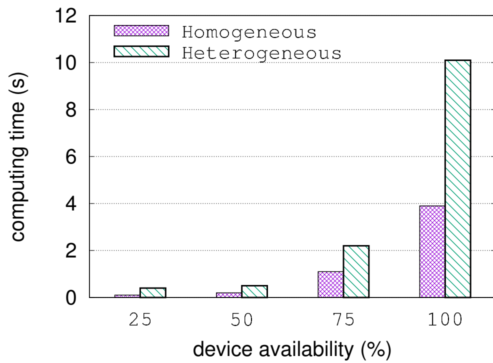

Fig. 7. Computing time under different ratios of online devices

To measure the latency overhead introduced by the usage of AWS technologies, we have also evaluated the time consumed at each stage of the processing flow and the obtained results are listed in Table I. From these results, we can tell that the latency overhead is negligible compared to AWS's benefits of auto-scalability and effortless manageability.

TABLE I
LATENCY OF CLOUD-FRAMEWORK

Core components involved	Time
User Interface to SQS queue	< 1 s
SQS queue to Lambda to Elastic beanstalk(On-demand)	$1 \sim 30$ s
Elastic beanstalk to Elastic compute cloud	< 1 s

B. Discussion about operational cost

Since iService is developed using AWS, monthly operational cost is inevitable. In order to lower the cost, different scales of AWS services can be used according to the estimated volume of IoT requests. Presumably, it would be sufficient to use lower computing resources when request volume is low (e.g., less than 3600 requests per hour) and appropriate to use larger computing resources when the volume is high (e.g., greater than 18000 requests per hour). Tables II) and III) compare the cost in these two cases. Apparently, the costs of EC2 instance and Elastic Beanstalk are much higher than other costs. Motivated by this observation, we have also conducted another set of experiments to evaluate whether the scheduler's actual performance is affected by the computing capability of EC2 instances. In these experiments, the computing time is evaluated under various number of requests by using both micro and large EC2 instances, and the obtained values are similar on both small and large scale EC2 instances. This result indicates that the request pattern, device availability and size of request group are dominant factor on the total time of service scheduling. Additionally, this result also suggests, without using an high-cost EC2 instance in practice, iService can still operate in an economical way while preserving a desired performance level.

TABLE II
AWS CLOUD COST - SMALL SCALE

AWS Technology	Cost
SQS	$1.00 per month ($0.50 per million requests)
Lambda	$0.40 per month ($0.20 per million requests)
Elastic Beanstalk	$33.40 per month (t2.medium EC2 instance)
EC2 instance	$33.40 per month (t2.medium EC2 instance)
RDS - MySQL Instance	$24.48 per month (db.t2.small)
Amazon S3	$0.021 per GB per month

TABLE III
AWS CLOUD COST - LARGE SCALE

AWS Component	Cost
SQS	$6.00 per month ($0.50 per million requests)
Lambda	$2.40 per month ($0.20 per million requests)
Elastic Beanstalk	$276.48 per month (m5.2xlarge EC2 instance)
EC2 instance	$276.48 per month (m5.2xlarge EC2 instance)
RDS - MySQL Instance	$97.92 per month (db.t2.large)
S3	$0.021 per GB per month

V. RELATED WORK

The cloud-based scheduling service is a novel design for scheduling IoT requests. However, there exists a large amount of research solving traditional scheduling problems. The closest work to the IoT service scheduling problem is the unrelated parallel machine (UPM) scheduling problem.

When solving the classic UPM scheduling problems, minimizing the total completion-time, the total tardiness and the total makespan of the jobs are the three major objective functions. Under these objective functions, there have been many heuristic algorithms proposed according to the rules listed in [11]. Some of the widely used scheduling heuristics are earliest due date (EDD), longest processing time (LPT), weighted shortest processing time (WSPT), first come first served (FCFS), most work remaining (MWKR) and least work remaining (LWKR). Though heuristic algorithms are more effective than pure optimization algorithms, the performance depends on two major factors, objective and problem model. When the classic UPM model does not well match a new problem, existing heuristic algorithms may be ineffective.

Since the job processing time is stochastic in nature, there are also plenty of research work focusing on stochastic model rather than dealing with classic, deterministic model. Shim et al. [12] addressed stochastic scheduling problem on unrelated machines by introducing time-indexed linear programming relaxation to reduce the total weighted completion time. Authors in [13] determined that combinatorial algorithm can be used to solve stochastic and non-preemptive scheduling problems to minimize the expectation of the total weighted completion time. In these works, there is only a single objective and the number of constraints is typically small. When the number of objectives or constraints increases, the comlexity of the problem also increases. In this case, genetic algorithms are one of the heuristic techniques that use random search to find the best heuristic solution. Researchers have used genetic algorithms to solve some of the scheduling problems such as the single machine unweighted tardiness problem [14]–[16], single machine scheduling subject to breakdowns [17] and classic job shop tardiness scheduling [18], [19]. However, there is no guarantee that genetic algorithms will provide an optimal solution even for small size problems.

Branch and Bound algorithm (B&B) is one of the most common methods to find optimal solutions for discrete and combinatorial problems. Multiple instances of the scheduling problem with different machines are generated and solved using B&B [20]–[22]. Authors in [8] formulated a zero-one mixed integer program (MIP) and developed B&B to minimize the makespan on unrelated parallel machines. Moreover, B&B is also guaranteed to find the best possible solution if it explores the full search space, and it's more efficient when the B&B algorithm can prune the search space in practice. In our solution, we solve the IoT service scheduling problem by using the Bonmin solver [10] which implements an efficient B&B algorithm.

VI. Conclusion

In this paper, we proposed a design of cloud-based online scheduling service, called iService, with the purpose of enhancing the quality of service in pervasive living spaces. Different from traditional scheduling problems, the core problem addressed in iService's design considers IoT-specific factors such as device location, sensing capability, and energy cost. Such a unique scheduling problem has been formulated as a mixed integer nonlinear programming (MINLP) problem. The proposed iService framework has been implemented and deployed to AWS, and its effectiveness have been demonstrated via extensive experiments under different request patterns and system parameters. Although iService is a novel approach for online scheduling of user requests on various IoT devices, there are several future directions worth exploration. In the current implementation, Bonmin is employed to solve the scheduling problem, but there are also several MINLP solvers such as BARON, AOA, and Knitro. As future work, a comparison study can be conducted in order to examine which one performs the best under the context of IoT service scheduling.

References

[1] I. Ceaparu, J. Lazar, K. Bessiere, J. Robinson, and B. Shneiderman, "Determining Causes and Severity of End-User Frustration," *International Journal of HumanComputer Interaction*, vol. 17, no. 3, pp. 333–356, Sep. 2004.

[2] I. Arapakis, X. Bai, and B. B. Cambazoglu, "Impact of response latency on user behavior in web search," in *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, 2014, pp. 103–112.

[3] J. Lazar, A. Jones, M. Hackley, and B. Shneiderman, "Severity and impact of computer user frustration: A comparison of student and workplace users," *Interacting with Computers*, vol. 18, no. 2, pp. 187–207, Mar. 2006.

[4] J. A. Jacko, A. Sears, and M. S. Borella, "The effect of network delay and media on user perceptions of web resources," *Behaviour & Information Technology*, vol. 19, no. 6, pp. 427–439, Jan. 2000.

[5] F. A. Chudak and D. B. Shmoys, "Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds," *Journal of Algorithms*, vol. 30, no. 2, pp. 323 – 343, 1999.

[6] J. M. Jaffe, "Efficient scheduling of tasks without full use of processor resources," *Theoretical Computer Science*, vol. 12, no. 1, pp. 1 – 17, 1980.

[7] R. Conway, . Miller, Louis W., and . Maxwell, William L., *Theory of scheduling*. Reading, Mass. : Addison-Wesley Pub. Co, 1967, bibliography: p. 249-258.

[8] O. Ozturk, M. A. Begen, and G. S. Zaric, "A branch and bound algorithm for scheduling unit size jobs on parallel batching machines to minimize makespan," *International Journal of Production Research*, vol. 55, no. 6, pp. 1815–1831, 2017.

[9] "Ampl — streamlined modeling for real optimization," https://ampl.com/products/ampl/ampl-for-research.

[10] "Basic open-source nonlinear mixed integer programming," https://www.coin-or.org/Bonmin/.

[11] S. S. Panwalkar and W. Iskander, "A survey of scheduling rules," *Operations Research*, vol. 25, no. 1, pp. 45–61, 1977.

[12] S.-O. Shim and Y.-D. Kim, "A branch and bound algorithm for an identical parallel machine scheduling problem with a job splitting property," *Computers & Operations Research*, vol. 35, no. 3, pp. 863 – 875, 2008.

[13] V. Gupta, B. Moseley, M. Uetz, and Q. Xie, "Stochastic online scheduling on unrelated machines," in *Integer Programming and Combinatorial Optimization*, F. Eisenbrand and J. Koenemann, Eds. Cham: Springer International Publishing, 2017, pp. 228–240.

[14] C. Dimopoulos and A. M. S. Zalzala, "A genetic programming heuristic for the one-machine total tardiness problem," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, vol. 3, July 1999, pp. 2207–2214.

[15] C. Dimopoulos and A. Zalzala, "Investigating the use of genetic programming for a classic one-machine scheduling problem," *Advances in Engineering Software*, vol. 32, no. 6, pp. 489–498, 6 2001.

[16] T. P. Adams, "Creation of simple, deadline, and priority scheduling algorithms using genetic programming," in *Genetic Algorithms and Genetic Programming at Stanford*, 2002.

[17] W.-J. Yin, M. Liu, and C. Wu, "Learning single-machine scheduling heuristics subject to machine breakdowns with genetic programming," in *The 2003 Congress on Evolutionary Computation*, vol. 2, Dec 2003, pp. 1050–1055.

[18] L. Atlan, J. Bonnet, and M. Naillon, "Learning distributed reactive strategies by genetic programming for the general job shop problem," in *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, 1994.

[19] K. Miyashita, "Job-shop scheduling with genetic programming," in *Proceedings of the 2Nd Annual Conference on Genetic and Evolutionary Computation*, 2000, pp. 505–512.

[20] E. Davis and J. M. Jaffe, "Algorithms for scheduling tasks on unrelated processors," *J. ACM*, vol. 28, no. 4, pp. 721–736, Oct. 1981.

[21] P. De and T. E. Morton, "Scheduling to minimize makespan on unequal parallel processors," *Decision Sciences*, vol. 11, no. 4, pp. 586–602, 1980.

[22] A. Hariri and C. Potts, "Heuristics for scheduling unrelated parallel machines," *Computers & Operations Research*, vol. 18, no. 3, pp. 323 – 331, 1991.