MINERVA: A Reinforcement Learning-based Technique for Optimal Scheduling and Bottleneck Detection in Distributed Factory Operations[1]

Tara Elizabeth Thomas Jinkyu Koo Somali Chaterji Saurabh Bagchi
School of Electrical and Computer Engineering
Purdue University
{thoma579, kooj, schaterji, sbagchi}@purdue.edu

***Abstract*— In manufacturing systems, the term bottleneck refers to a component that limits the entire throughput of a system. A number of approaches have attempted bottleneck detection. However, existing solutions have limitations, leaving the bottleneck identification still no trivial task. To address the shortcomings of prior works, we study Job Shop Scheduling Problems (JSSP) with the realistic extension that jobs are enqueued periodically, and propose a machine learning based solution to such a problem, named MINERVA. MINERVA first finds the optimal resource scheduling for a target interval, based on a model-free reinforcement learning technique. Then, using an artificial neural network classifier, MINERVA identifies the constrained resources for each target interval. MINERVA is evaluated on two representative benchmarks with the key result being that MINERVA is able to detect the system bottleneck(s) with a high accuracy of 95.2%, which is almost 25% better than the best-in-class bottleneck identification methods.**

I. INTRODUCTION

The advancements in miniaturized and communication-enabled sensors and computer software for them are causing significant changes in the field of factory operations and manufacturing. With the increasing scale of factory operations and the large amounts of real-time data available therefrom from various embedded sensors, we can now optimize factory operations in a data-driven manner. A logical and cost-effective way to come up with the desirable design points is to use simulation-based modeling of factory operations [28]. System simulations are important tools used in the modeling of factory performance because such tools allow the associated personnel to finetune the input parameters without realtime changes in the system configurations, which may well be untenable in a production environment. Finding an optimal resource allocation and scheduling policy for the job shop scheduling problem (JSSP), also known as the job-shop problem, is a non trivial optimization problem in computer science and operations research in which ideal jobs are assigned to the ideal resources at the correct times within the ambit of the smart factory or any other distributed system for that matter [20]. These distributed systems often have bottlenecks that prevent them from reaching optimal performance. A bottleneck is defined simply as a machine (resource) in the system that limits the job's throughput such that increasing the capacity or availability of the bottlenecked resource increases performance. It is crucial to identify and eliminate these bottlenecks to achieve the best possible performance [19], [34]. In many cases, these bottlenecks are complex and dynamic *i.e.,* they vary with time and with other system parameters like the job distribution [3].

In this paper, we solve the two related problems in the context of a distributed factory operation with multiple sensor-equipped machines: optimal scheduling of arriving jobs to machines and bottleneck identification (under the optimal scheduling policy) to improve the throughput till the desired throughput is achieved.

The classic JSSP aims to assign a given number of job types to a given set of *machine pools* (here, *resource pools*) so that some objective function is optimized and is a well-traversed optimization problem in computer science and operations research [8], [36], [14]. The above papers have presented a variety of solution approaches, which we analyze in detail in Section VII. Fundamentally, these JSSP models have a *fixed number* of job types and fixed number of resource pools. Each resource pool consists of some number of identical machines, which is referred to as the capacity of the resource pool. After a job has been scheduled and started to be processed on a machine, the machine can process another job only after the completion of the current one, *i.e.,* there is no preemption. Each job type has a fixed sequence of resource pools that it has to be processed on. Each machine is able to process only one job at a time. However, this classic JSSP is a much simpler version of the real world of factory operations. In a more realistic scenario, there is a continuous stream of jobs arriving to be processed rather than a fixed number of jobs under each job type [7]. This indicates that there is a change in the state of the system as time progresses. So, there is stochasticity in the arrival rate of the jobs, and hence the system is not completely known at the beginning of the operation, as all prior work assumes. So, the scheduling policy, which is essentially a rule that governs the job assignment to the available resource pools, needs to be flexible, possibly a function of the current state, which can be modeled as a Markov process.

The problem of finding the optimal scheduling policy can be effectively solved by using reinforcement learning (RL). This is because for any system that can be modeled

[1]This work is supported by the National Science Foundation under Grant Numbers CNS-CNS-1527262 and CNS-1513197 and by General Electric Corporation through the PRIAM center at Purdue. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the sponsors' views.

978-1-5386-1182-1/18/$31.00 ©2018 IEEE

as a Markov decision process, RL can enable the learning process of the software agent, in a statistical sense, based on the feedback it gets when it performs different actions with changes in ambient signals. In the case of this problem, since our objective is to maximize the throughput, when the agent makes a good scheduling decision, the throughput increases, and hence it gets a positive reward, which can further reinforce this action. In contrary, if the agent takes an action that is a bad scheduling decision, the throughput decreases, resulting in a penalty that discourages the agent from taking such decisions in the future. This technique is also generic in the sense that it can be applied to any JSSP model or smart-factory model because RL starts out by treating its environment as a black box, and learns about the environment as time progresses, through the feedback process. Another reason why RL is a good fit for our problem is because it can focus on maximizing the cumulative long term reward as opposed to getting stuck in short-term local maxima in the throughput. This means that the agent will be able to learn a scheduling policy that will result in greater throughput in the longer term (globally), notwithstanding local drops in throughput. Some prior works have successfully used reinforcement learning algorithms to solve JSSP [30], [5], [35], [37]. However, none of them is applicable to the continuous arrival of jobs to the system, as they all assume that the number of jobs is discrete and a constant number. Moreover, none of them has addressed the problem of finding the bottleneck resource in the system.

In this paper, we present a new technique called MINERVA, which solves two problems in a distributed workflow representative of a factory floor—first, it creates an optimal schedule of the jobs on the resource pools for a continuously arriving set of jobs, and second, it identifies which machine pool is the bottleneck and relieves the bottleneck by adding capacity. A schematic of MINERVA is given in Figure 1. MINERVA first optimizes the scheduling of the various jobs within the factory model using reinforcement learning. It learns the scheduling actions that would lead to maximization of cumulative rewards using Q-learning, increasing system throughput. Since the state space of a realistic factory model is huge (it grows exponentially with the numbers of jobs currently active in the system and the number of different types of machine pools), it is impractical to use simple Q-learning as it would take too much time to converge over the millions of iterations needed to cover all these states, multiple times, in order to learn the optimal policy. So, MINERVA uses an approximate Q-learning method that uses neural networks to approximate the Q-function, proven to work well on real-time systems with huge state spaces [26], [27]. If the throughput with the optimal schedule is still less than the desired throughput, MINERVA finds the bottleneck machine pool in the system, so that its capacity could be increased for further increase in the throughput. MINERVA leverages some of the bottleneck identification metrics already existing in literature, such as the length of the queue at each machine pool, which can be observed from the factory model. It then applies a previously learned neural network to identify the system bottleneck based on these metrics. Once the bottleneck resource pool is identified, its capacity is increased by 1 unit, mimicking the addition of a machine to that machine pool. This updated system is then simulated and the above steps are repeated until the desired throughput is reached.

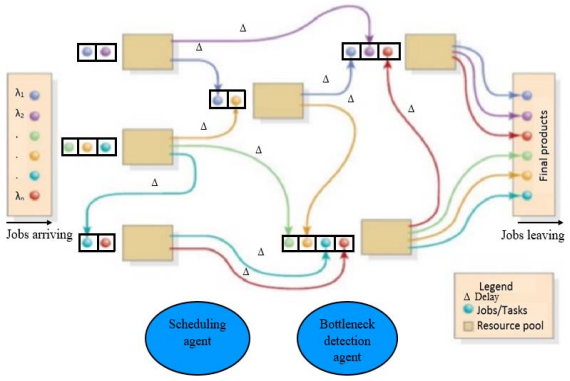

Fig. 1. Schematic of MINERVA

We consider a physically distributed set of resources and jobs that have to be physically routed among them. Thus, there are physical transfer times as well as digital communication times involved for the metrics to reach the central scheduler, such as, queue length at any resource, and for the scheduler to dispatch its bottleneck mitigation command. We incorporate these delays into our formulation and solution and study their effects on the overall throughput.

This paper makes the following key contributions:

1) We develop MINERVA, an approximate reinforcement learning (RL)-based technique to ensure optimal scheduling for the dynamic job shop scheduling problem (JSSP). We demonstrate how effectively neural networks can be used to approximate the Q-functions for these problems, which have a huge state space.
2) We develop a neural network-based bottleneck identification technique to identify and eliminate system bottlenecks that limit throughput in distributed systems.
3) We implement the above mentioned techniques on representative benchmarks, with the added realistic extension to be applicable to a continuous and dynamic inflow of jobs. We show that MINERVA performs much better than the best-in-class techniques.

II. BACKGROUND

A. JSSP and scheduling

Classic JSSP [36] has been an important research area in both industrial engineering and operations research for half a century. The goal is to allocate a specified number of job types to a limited number of resource pools in such a way that some job-specific objective is optimized. A job-shop has n different possible job types, $J_1, J_2 \ldots J_n$, and m different resource pools $P_1, P_2 \ldots P_m$. Each job within a job type has a particular fixed, sequence of operations to be performed on it to transform the input into the final

desired product. A particular operation on a job can be performed by machines of a particular resource pool, and takes a fixed amount of processing time to complete. A job is finished after completion of its last operation. Typically, the number of resource pools is less than or equal to the number of job types. After a job has been scheduled and started to be processed on a machine, the machine can process another job only after the completion of the current one. The desired optimization in JSSP is usually the minimization of makespan or the total length of the schedule when all the jobs in the systems have been completed. One of the examples of a classic job shop problem, usually referred to as ft06 [4] and its solution [16] is given in Figure 2 for a better understanding. Each row in the table represents a job, and the columns have the machine numbers and the processing times for the different operations on the job. The operations have to be performed in a specific sequence. For job 1, it has to be processed first on machine 3 and that takes 1 time unit. Then, it has to be processed on machine 1 and that takes 3 time units, and then, on machine 2 for 6 time units, and so on. Each job goes through a 6-stage pipeline. For this paper, we consider the more realistic extension of JSSP, whereby jobs keep coming in with some stochastic distribution, and are enqueued for the resource pools they need. The resource pools could have more than a single machine each, as opposed to the classic JSSP model. The objective is maximize the system's throughput.

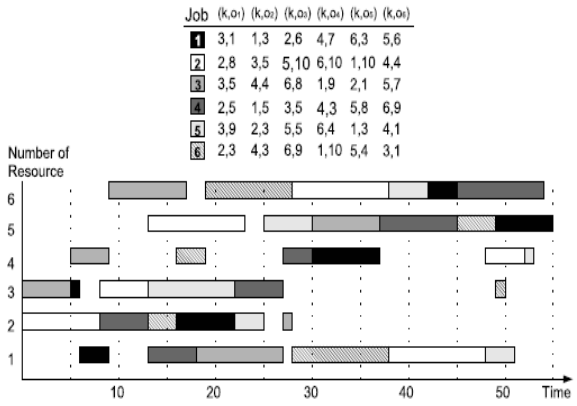

Fig. 2. The FT06 JSSP and a Gantt chart representing its optimal solution.

B. Reinforcement learning

RL is an area of machine learning inspired by behavioral psychology that aims at making a software agent learn the optimal action for each system state, based on the rewards or penalties ensuing from its actions. This is an automated, non-supervised learning technique where the agent, which initially does not have any knowledge of its environment, starts out by taking random actions. Then, it iteratively takes different actions that might change the state of the agent, and might also give it a positive or negative reward, as feedback, depending on whether the result of the action is favorable or unfavorable. Based on this reward, the agent eventually learns which actions are optimal for each state. The optimal action would be the one that maximizes the agent's expected long-term reward. A simple example to understand this would be a software agent playing the video game, Pacman, which earns points on eating pellets and dies on meeting a ghost. The state of the agent would have information like positions of Pacman, pellets, and ghosts. The possible actions are staying still, or moving to the left, right, up, or down. The rewards would be positive when the agent eats the pellets, and negative when the agent does something unfavorable, like meeting a ghost.

Q-learning is a model-free RL technique that can find the optimal action-selection policy for any given Markov Decision Process. It works by learning an action-value function, called the *Q*-function, which eventually gives the expected utility of taking a particular action in a particular state and follows the optimal policy thereafter. The core of the learning is a simple iteration process based on equation below.

$$(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha\left(r_t + \gamma \max_{a_{t'}} Q(s_{t'}, a_{t'})\right) \tag{1}$$

where s_t, a_t, r_t are the state, action and reward at the t time. Here, $\alpha < 1$ is called the learning rate, denoting the extent to which the current observation affects the Q-value, and γ is a discounting factor of future rewards.

In realistic cases of the distributed factory operations that we observed, the possible state space is huge, so the canonical *Q*-learning algorithm would not converge in any reasonable amount of time as it would have to learn the best action for each of these states, separately, to reach an optimal policy. The number of iterations required for this is extremely high and we need to apply approximate *Q*-learning. In such scenarios, approximate *Q*-learning techniques, where the *Q*-function is approximated by decision trees or neural networks is used. Using deep neural networks as *Q*-function approximators has been proven efficient in various applications, including Google's AlphaGo [26], [27].

III. System Model

We consider a factory model where many jobs (tasks) of different types are to be processed on different machines (resources). Jobs of n different job types $J_1, J_2 \ldots J_n$ enter the system, each with Poisson distributed arrival rates $\lambda_1, \lambda_2, \ldots, \lambda_n$. The factory has m sets of different resource pools: $P_1, P_2, \ldots P_m$. Each resource pool P_i has c_i identical resources/machines that can perform a fixed set of operations. Every machine can be either 'busy', which means it is currently performing an operation on a job, or 'idle', which means it is not performing any operations on jobs at the time. The c_i is called the capacity of the resource pool P_i. Because of practical cost constraints in factory setups, the total possible capacities of all resource pools is limited to the maximum capacity $\bar{c}_i$.

Each job type has a fixed set of operations to be performed on it sequentially, to convert it to the final product. Each of these operations has two values associated with it: the machine on which the operation can be performed, and

the processing time needed for the operation to complete. For example, operation $o_1(3,1)$ means that operation o_1 requires a machine from resource pool P_3 for 1 time unit. We consider only deterministic processing times in this paper. It is important to note that different job types require different operational sequences to be performed on them. The resources are physically distributed, and thus, there is a delay for any job to travel from one machine to another. We consider this delay as an experimental parameter in our simulations. There is a desired throughput for this factory model and our goal is to achieve it by both *optimal scheduling* and *proper bottleneck identification and elimination.*

For this purpose, we consider the factory model as a single Markov decision process (MDP) where the state of the system at any time t, s_t is comprised of the information about the processing status of different jobs and resource pools in the system. This can be represented as $s_t = (s^i_{t,1}, \ldots, s^i_{t,m}, s^o_{t,11}, \ldots, s^o_{t,nm}, s^c_{t,1}, \ldots, s^c_{t,m}, s^h_t)$, where $s^i_{t,k}$ is 1 if there is an idle machine in the k-th resource pool and 0 if all machines in the k-th resource pool are busy; $s^o_{t,lk} \in [0,1]$ is an indicator of number of jobs of job type l waiting for k-th resource pool (L_{lk}). The $s^o_{t,lk}$ is 1 if L_{lk} is greater than a threshold value $\bar{L}_k$, and $s^o_{t,lk} = L_{kl}/\bar{L}_k$ otherwise; $s^c_{t,k} \in (0,1)$ where $s^c_{t,k}$ is $c_k/\bar{c}_k$ and s^h_t is a ratio of the current throughput to the desired throughput. Furthermore, the terminal state (s_T) occurs when the system has been simulated for the time interval of our concern (T).

To detect the bottleneck resource pools in the system, the model also collects bottleneck metrics like the average waiting time (W_i), the average queue length (L_i), and average utilization of each (U_i) resource pool over each time interval. The information $bnData = \{W_i, L_i, U_i$ and c_i for each $P_i\}$ is stored for each time interval over which the bottleneck has to be detected.

IV. Design

A. The machine learning based decision making agents

MINERVA has two machine learning based decision making agents: the scheduling agent and the bottleneck detection agent. The first one can be referred to as the inner loop decision making agent and takes scheduling decisions. This agent comes into play at times t when either of the following events of scheduling interest happen: when a resource becomes idle, or when a new job arrives to be processed by an idle machine that has no other jobs waiting for it. It decides which among the jobs waiting for a resource to pick. The resource then starts processing that job and the simulation continues until another resource or even the same resource becomes idle again. Based on whether the schedule picked was good (leads to more throughput) or bad, the simulation model provides a feedback to the decision making agent, which uses this to refine decisions in future iterations. This agent is essentially a reinforcement learning agent that learns the optimal scheduling action for each system state, over time. The possible actions $a_t \in A$ taken by the scheduling RL agent, at any instant of time t, describes the selected job type to be scheduled and the resource pool on which it is to be scheduled. The complete set of all possible actions is $A = \{a_{jp}\}$, where a_{jp} is the action of scheduling a job type j on resource pool p. Though this complete set of possible actions is fixed, the set of actions possible at any particular time varies dynamically because not all resource pools will have a free machine at that instant, and neither will all the different job types be waiting for a particular resource pool at that time. This is different from typical RL problems where the action space is fixed. We developed our RL algorithm such that it can handle a dynamic action space by choosing only from the set of possible actions, which will be communicated to the agent from the simulation model. The overall aim of the scheduling RL agent is to find a scheduling policy π that maximizes the accumulated reward over time. The reward for the agent is dependent on the throughput of the system, which is the objective to be maximized in our problem. For each action a_t that results in the next state $s_{t'}$, we give a small reward proportional to the throughput. For the final state, if the throughput is greater than a threshold, we give a big positive reward, and if the throughput is lower than the threshold, we give a penalty. This can be represented as

$$r_t = \begin{cases} k_1 \cdot H_t \text{ if } s_t \neq s_T \\ k_2 \text{ if } s_t = s_T \text{ and } H_t \geq H \\ -k_2 \text{ if } s_t = s_T \text{and } H_t \leq H \end{cases}$$

where k_1 and k_2 are all positive numbers ($k_2 >> k_1$), H_t is the throughput at time t, and H denotes the desired throughput. Once the simulation has been iterated enough, the *Q*-learning cost function becomes minimal and the policy converges. This means that the agent has learned an acceptable schedule, the training phase has been completed, and the agent continues to schedule using this learned policy. MINERVA also finds the bottleneck in the system at regular intervals of time, if the throughput is less than the desired throughput using the bottleneck detection agent—a neural network-based classifier. Any one machine among the resource pools can be the system bottleneck at any given time. The agent is initially trained using *Algorithm 1* in IV-B. At periodic intervals, this trained agent looks at the bottleneck metrics ($bnData$) of the system for that interval and outputs the bottleneck resource pool P_{bn} in the system. Then, it increases the capacity of the bottleneck resource pool by one unit to increase the throughput of the system and continues to run the simulation. The possible outputs for the bottleneck prediction agent, always an integer from 0 to m, denotes one of the resource pools as bottleneck (1 to m) or no bottleneck in the system (0).

B. The Learning Algorithm

MINERVA uses an approximate *Q*-learning technique, where neural networks are used to approximate the *Q*-function for implementing the scheduling agent. We use the ϵ greedy strategy to ensure that exploration of the state space is given priority initially, and once it has learned sufficiently about the state space, it tries to reduce random

exploration and resorts to rewarding actions. ϵ denotes the extent of exploration to be done- $\epsilon = 1$ means that the agent always takes random actions for maximum exploration, and $\epsilon = 0$ means that the agent always follows the currently known optimal policy with no random actions. The ϵ greedy strategy works by having a high value for ϵ initially, so that it gets to explore the state space, and gradually reducing the ϵ value over time. Every simulation run by the agent is called an episode. MINERVA also utilizes a technique called experience replay [23], where a set consisting of the state, action, next state, and the reward, which we call 'experience' at each time-step, $e_t = (s_t, a_t, r_t, s_{t'})$ is collected over many episodes into a replay memory data-set $D = e_1 \dots e_N$, of fixed size. The algorithm used is *Algorithm 2*. A minibatch of random experience samples $e \in D$ are taken from the replay memory with all the experience samples. *Q*-learning updates as in II-B are performed on this minibatch of samples. This leads to updated *Q*-function. After performing this experience replay, the agent selects and executes an action according to an ϵ greedy policy. Then, the reward is obtained and this experience is added to the replay memory. These steps are repeated multiple times until a convergent policy is obtained. Since each experience entry is used in many weight updates during the learning process, it results in higher data efficiency. If online policies are used, there is a possibility of undesirable feedback loops arising during the learning, which could lead to the parameters getting stuck in local minima, or even diverging hugely [33]. By using experience replay, the behavior distribution is averaged over many of its prior states, smoothing out learning and avoiding oscillations that would delay convergence. In our prior work [18], [17], we have had to do feature engineering for obtaining good classification, but here the problem space is simpler and all features are real-valued and we empirically did not find a need for such feature engineering.

Algorithm 1 Learning to identify bottleneck resources

procedure VERIFY TRUE BOTTLENECK(seed)
  Run the AnyLogic system model for chosen seed.
  Note $bnData = \{W_i, L_i, U_i$ and c_i for each $P_i.\}$
  for $i \leq numResourcePools$ **do**
    Increase capacity of P_i by 1.
    Run the AnyLogic model and note throughput.
  end for
  Mark P_i whose increase in capacity led to maximum throughput as the true bottleneck (P_{bn}).
  Return the $bnData$ and P_{bn}.
end procedure
procedure LEARN BOTTLENECK IDENTIFICATION
  repeat
    Choose a random seed, with random model parameters.
    Run procedure VERIFY TRUE BOTTLENECK.
    Store the returned $bnData$ and P_{bn} as an entry in the data set for supervised learning.
    $dataSetSize \leftarrow dataSetSize + 1$
  until $dataSetSize = D_{ss}$
  Separate out the collected data into disjoint training data ($data_{tr}$), validation data ($data_{vl}$) and testing data($data_{ts}$).
  Use $data_{tr}$ to train a neural network to detect P_{bn} from $bnData$.
  Cross validate the neural network using the $data_{vl}$.
  Test the accuracy of the neural network using $data_{ts}$.
end procedure

Algorithm 2 Learning the optimal scheduling

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode = $1 : M$ **do**
  Initialize system state s_1.
  for $t \in (1, T)$ when an event of scheduling interest happens **do**
    With probability ϵ select a random action a_t. Each action a_t represents processing particular job on a particular machine
    Otherwise select $a_t = \arg\max_a Q(s_t, a)$
    Execute action a_t in the AnyLogic jobshop model.
    Store transition (s_t) in D
    Sample a minibatch of transitions $(s_t, a_t, r_t, s_{t'})$ from D
    if $s_{t'}$ is a terminal state **then**
      Set: $y_t \leftarrow r_t$
    else
      Set $y_t \leftarrow r_t + max_a Q(s_{t'}; a)$
    end if
    Perform a gradient descent step on $(y_t - Q(s_t, a_t))^2$
  end for
end for

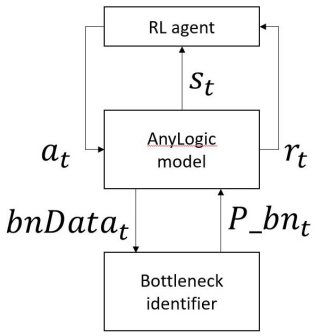

Fig. 3. Overall workflow of MINERVA

The overall workflow of MINERVA is in Figure 3. While the scheduling agent is called continuously, the bottleneck-identification agent is called only once per fixed time interval.

V. IMPLEMENTATION

The factory models to be optimized were created as discrete event simulation models in AnyLogic 7 [1], a multimethod Java-based simulation software. The RL agent modeling was done using the Java-based deep learning library *dl4j* [2] for tight integration with AnyLogic. The agents described in IV-A as well as the simulation model explained above are part of the top-level AnyLogic experiment. The simulation model implements an MDP interface that provides functions for the agents to easily start a new instance of the simulation model, to pause, run, and reset a simulation. This interface also provides the agents the encapsulated information about the current simulation system state, the possible action space, the reward, an indication of the simulation termination, and a way for the agents to communicate the next action to be taken to the model.

VI. EVALUATION AND RESULTS

We used an extension of the ft06 problem, as described in Section III, to evaluate MINERVA. The model was created in AnyLogic as a discrete-event simulation model. The scheduling agent was integrated into the model using the MDP

interface and was run with the objective of finding an optimal schedule. It was observed that the agent found a schedule that gave an average throughput of 56 jobs/time unit, which is significantly higher than throughputs using the baseline techniques—FIFO and Shortest Processing Time (SPT)—the most common industrial dispatching rules. MINERVA gets this improvement because it uses the relevant information like the queue length for each operation and the number of resources in each pool from the simulation model to make a scheduling decision, rather than just naively following a fixed rule. Figure 4 summarizes this result. We evaluated

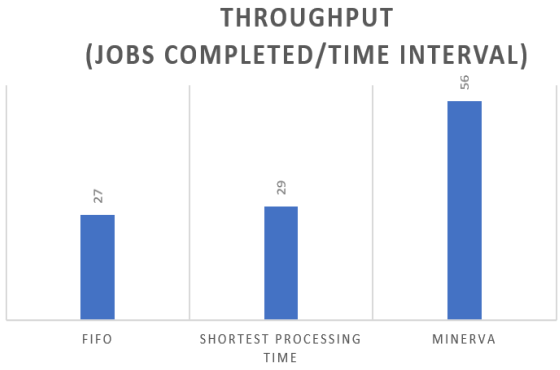

Fig. 4. Throughput by MINERVA scheduling *vs.* popular scheduling rules

the bottleneck identification agent, detailed in IV, on the realistic extension of the classic FT06 JSSP with continuous job arrivals. It was seen that MINERVA was able to identify system bottlenecks with a much higher accuracy (92.6%) when compared to other traditional techniques, all with an accuracy of less than 75% on the same test set, as shown in Figure 4. This is because one bottleneck metric alone is not really sufficient to detect the bottleneck. For instance, if a resource pool has a long queue length, but if all the jobs waiting in the queue have a very short processing time, this resource pool might not actually be a bottleneck. Or, for example, the average waiting time of a resource pool could have been high because of some particular job that takes a long processing time, and might not be really due to the resource pool posing a bottleneck. MINERVA performs better than these individual methods because it looks at a more holistic picture of the system states than what is portrayed by one bottleneck metric. However, MINERVA still fails when it is not able to judge the bottleneck resource pool with certainty based on the information it has. To get a better idea of the performance of MINERVA with respect to the other bottleneck methods, we used another benchmark model from [3], which we refer to as the *AD05 benchmark*. This benchmark model has four resource pools and five job types. The benchmark is described in Figure 6. This indicates that Job 1 had to be processed on resource pool 1 for 2 time units, then on resource pool 4 for 3 time units etc. The extended model with continuous job arrival has a stochastic Poisson distributed arrival rate of 1 job per 3 time units. It was seen that MINERVA was able to identify bottlenecks to an accuracy

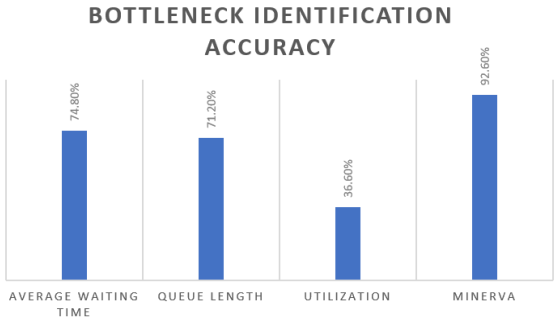

Fig. 5. Accuracy of bottleneck detection by MINERVA *vs.* popular methods for FT06 benchmark extension

of 95.6%, which is over 25% better than the accuracy of the best one among other methods (Figure 7).

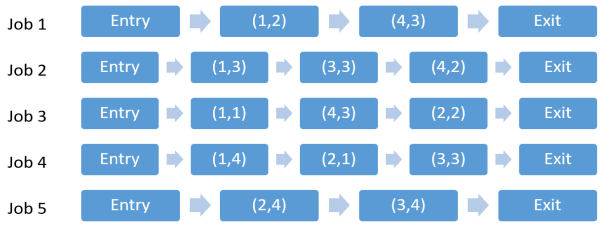

Fig. 6. Figure representing the AD05 JSSP benchmark

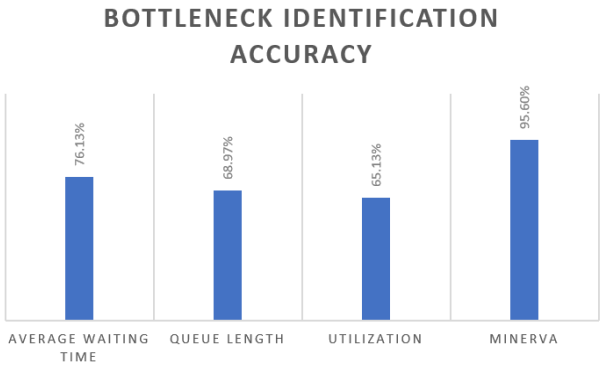

Fig. 7. Comparison of accuracy of bottleneck detection by MINERVA compared to bottleneck detection methods for extension of AD05 benchmark

We also studied the variation of throughput with increase in resource pool capacity for the two benchmarks. This was done so as to get a feel of how effective each bottleneck identification method is with respect to the increase in throughput. We used each of the different bottleneck identification methods—MINERVA, average queue length, utilization, and average waiting time, in order to detect the bottlenecks separately. The capacity of the identified bottleneck resource is then increased by 1 unit. The simulation is then continued for the same interval of time, and this bottleneck identification and elimination step is repeated for 4 more steps. Then, we plot the throughput *vs.* increase in capacity for each of the 4 bottleneck identification methods. The graphs are shown in

Figures 8 and 9. It is seen that MINERVA performs the best because it detects true bottlenecks with greater accuracy than the other methods. This means that with MINERVA, when the actual bottleneck resource pool's capacity is increased, the greatest increase in throughput can be obtained.

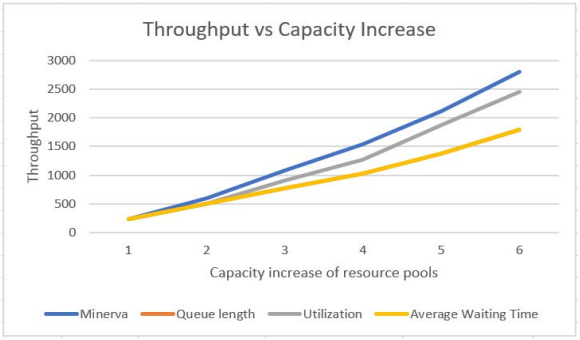

Fig. 8. Throughput *vs.* increase in resource pool capacity as identified by various bottleneck identification methods for extension of FT06 benchmark

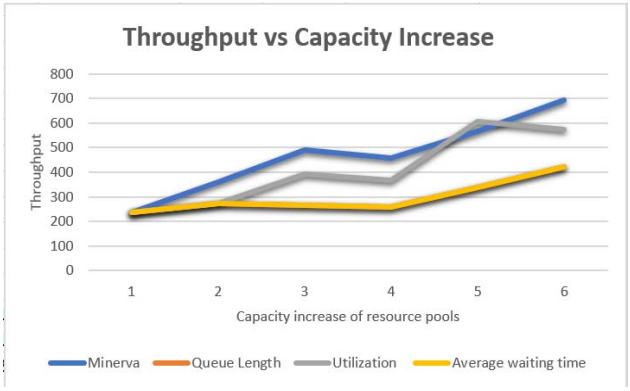

Fig. 9. Throughput *vs.* increase in resource pool capacity as identified by other bottleneck detection methods for AD05 benchmark extension

In reality, there would be some time delay in the machines communicating information to the centralized agent and for the agent to communicate the bottleneck mitigation action to the machines. There will also be delays in physically transferring jobs to the resource pools. However, the above experiments did not consider these delays. We performed an experiment where we included this delay in the simulation model and varied its value within a reasonable range. For both benchmarks, we vary this delay from 0 to 1 time unit. The relevant graph is in Figure 10. It is seen that there is a linear decrease of throughput with respect to the latency and hence it is of utmost importance to take necessary measures to reduce it. The decrease in throughput with increase in latency is steeper for AD05 benchmark as compared to FT06 benchmark. This is because, in general, the processing times of operations in the AD05 benchmark is lower compared to that in FT06 benchmark. This means that the latency to total processing time ratio of most jobs would be higher in AD05 than FT06, and hence, the effect of increased latency is more prominent in AD05. However, if we were to parallelize our entire process, the effect of the latency will be less prominent. Our problem though is *not* embarrassingly parallel and we will have to identify more subtle forms of extracting concurrency as we have done in [24] for example in the domain of computational genomics.

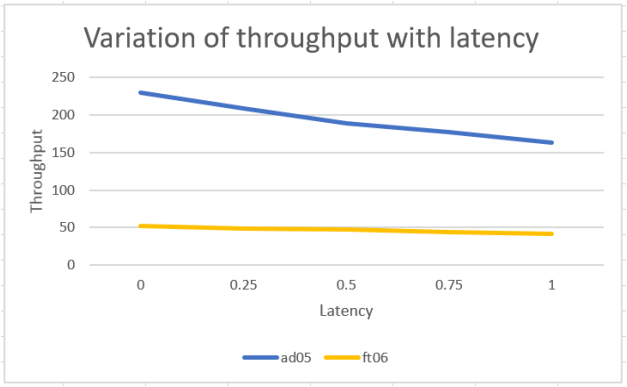

Fig. 10. Throughput *vs.* latency for FT06 and AD05 benchmarks' extension

VII. RELATED WORK

In industrial settings, it is important to eliminate bottlenecks if possible, such as through reprogramming parts of the pipeline [6]), to make the system as efficient as possible [19], [34]. Traditionally, utilization-based methods [20] have been used to identify bottleneck machines, whereby the machine with the highest utilization is considered to be the system bottleneck. Other commonly used bottleneck identification metrics are the queue length of jobs waiting at each machine [29] and the average waiting time for the machines [20]. There have been many system theoretical approaches to bottleneck identification over the years. Sengupta *et al.* [31] proposed to analyze the inter-departure time of the different machines in the system to identify the bottlenecks. Chiang *et al.* [13] suggested using frequencies of machine blockages and starvations as indicators of bottlenecks. The blockage and starvation probability of the machines are used to identify bottlenecks in [21]. However, these approaches are based on flowshop-like models, where there are machines that perform consecutive tasks arranged with buffers in between. Hence, they need information related to the structure of the factory system, which are generally not fixed in the case of job shops, as the sequence of machines depends on the job that is being processed. Also, several other proposed techniques like maximum average per hop delay [15] and workload matrix-based convex analysis [10] make specific assumptions like M/M/1 system and a closed queuing network respectively and are not applicable to generic factory models and job shops. An orthogonal dimension is how to measure the input metrics that will feed into the various types of models. There is a rich literature set in the area of monitoring operational systems, through software add-ons [12], [11] or through specialized hardware working with the software add-ons [32], [22].

VIII. CONCLUSION

This paper introduces MINERVA, an optimization technique to improve factory performance using approximate Q-learning aimed at job shop scheduling and prediction of

system resource bottlenecks. MINERVA is implemented on realistic extensions of representative classic JSSP benchmarks. MINERVA is also benchmarked against other competitive techniques, identifying bottlenecks with an accuracy of 95.6%, which is over 25% better than the best-in-class, increasing throughput. The effect of latency on the performance of the system was also studied. Future extension of this work would consider jobs with stochastic processing times, unlike in our current study where all jobs have deterministic times. Additionally, we will consider distributed processing of the two stages—optimal scheduling and bottleneck detection. Distributed processing will make the overall solution more scalable and thus applicable to larger problem sizes and to latency-sensitive systems, while trading off some accuracy. Finally, from a usability standpoint, we are considering developing a Domain Specific Language (DSL) that a system owner can use to describe the processing pipelines. There is rich history of such DSLs for various domains, such as Sarvavid [25] (computational genomics) or Delite [9] (for building DSLs).

REFERENCES

[1] AnyLogic: Multimethod Simulation Software and Solutions. https://www.anylogic.com/.

[2] Deep Learning for Java. https://deeplearning4j.org/.

[3] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management science*, 34(3):391–401, 1988.

[4] N. I. Anuar and A. Saptari. Performance evaluation of different types of particle representation procedures of particle swarm optimization in job-shop scheduling problems. In *IOP Materials Science and Engineering*. IOP Publishing, 2016.

[5] M. E. Aydin and E. Öztemel. Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, 33(2):169–178, 2000.

[6] S. Bagchi, N. B. Shroff, I. M. Khalil, R. K. Panta, M. D. Krasniewski, and J. V. Krogmeier. Protocol for secure and energy-efficient reprogramming of wireless multi-hop sensor networks, Jan. 31 2012. US Patent 8,107,397.

[7] C. Bierwirth and D. C. Mattfeld. Production scheduling and rescheduling with genetic algorithms. *Evolutionary computation*, 7(1):1–17, 1999.

[8] J. Błażewicz, W. Domschke, and E. Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European journal of operational research*, 93(1):1–33, 1996.

[9] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 89–100. IEEE, 2011.

[10] G. Casale and G. Serazzi. Bottlenecks identification in multiclass queueing networks using convex polytopes. In *MASCOTS*, pages 223–230. IEEE, 2004.

[11] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):300–301, 2003.

[12] A. Charapko, A. Ailijiang, M. Demirbas, and S. Kulkarni. Retrospective lightweight distributed snapshots using loosely synchronized clocks. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 2061–2066. IEEE, 2017.

[13] S.-Y. Chiang, C.-T. Kuo, J.-T. Lim, and S. Meerkov. Improvability of assembly systems i: Problem formulation and performance evaluation. *Mathematical Problems in Engineering*, 6(4):321–357, 2000.

[14] L. Davis. Job shop scheduling with genetic algorithms. In *Proceedings of an international conference on genetic algorithms and their applications*, volume 140, 1985.

[15] G. F. Elmasry and C. J. McCann. Bottleneck discovery in large-scale networks based on the expected value of per-hop delay. In *IEEE MILCOM*, volume 1, pages 405–410. IEEE, 2003.

[16] T. Gabel and M. Riedmiller. Adaptive reactive job-shop scheduling with reinforcement learning agents. *International Journal of Information Technology and Intelligent Computing*, 24(4), 2008.

[17] A. Ghoshal, A. Grama, S. Bagchi, and S. Chaterji. An ensemble svm model for the accurate prediction of non-canonical microrna targets. In *ACM-BCB Best Paper Award*, pages pp–403. ACM, 2015.

[18] A. Ghoshal, R. Shankar, S. Bagchi, A. Grama, and S. Chaterji. Microrna target prediction using thermodynamic and sequence curves. *BMC Genomics*, 2015.

[19] I. Laguna, S. Mitra, F. A. Arshad, N. Theera-Ampornpunt, Z. Zhu, S. Bagchi, S. P. Midkiff, M. Kistler, and A. Gheith. Automatic problem localization via multi-dimensional metric profiling. In *32nd IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 121–132. IEEE, 2013.

[20] A. M. Law, W. D. Kelton, and W. D. Kelton. *Simulation modeling and analysis*, volume 2. McGraw-Hill New York, 1991.

[21] L. Li. Bottleneck detection of complex manufacturing systems using a data-driven method. *International Journal of Production Research*, 47(24):6929–6940, 2009.

[22] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *Proceedings of the 12th international conference on Information processing in sensor networks*, pages 153–166. ACM, 2013.

[23] L.-J. Lin. Reinforcement learning for robots using neural networks. Technical report, CMU, 1993.

[24] K. Mahadik, S. Chaterji, B. Zhou, M. Kulkarni, and S. Bagchi. Orion: Scaling genomic sequence matching with fine-grained parallelization. In *Supercomputing 2014 (The International Conference for High Peformance Computing, Networking, Storage and Analysis)*, pages pp–1. IEEE, 2014.

[25] K. Mahadik, C. Wright, J. Zhang, M. Kulkarni, S. Bagchi, and S. Chaterji. Sarvavid: A domain specific language for developing scalable computational genomics applications. In *Proceedings of the 2016 International Conference on Supercomputing*, page 34. ACM, 2016.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[28] D. Mourtzis, N. Papakostas, D. Mavrikios, S. Makris, and K. Alexopoulos. The role of simulation in digital manufacturing: applications and outlook. *International journal of computer integrated manufacturing*, 28(1):3–24, 2015.

[29] P. Pollett. Modelling congestion in closed queueing networks. *International Transactions in Operational Research*, 7(4-5):319–330, 2000.

[30] S. Riedmiller and M. Riedmiller. A neural reinforcement learning approach to learn local dispatching policies in production scheduling. In *IJCAI*, volume 2, pages 764–771, 1999.

[31] S. Sengupta, K. Das, and R. P. VanTil. A new method for bottleneck detection. In *Proceedings of the 40th conference on Winter simulation*, pages 1741–1745, 2008.

[32] M. Tancreti, M. S. Hossain, S. Bagchi, and V. Raghunathan. Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 288–301. ACM, 2011.

[33] J. N. Tsitsiklis and B. Van Roy. Analysis of temporal-diffference learning with function approximation. In *NIPS*, pages 1075–1081, 1997.

[34] Y. Wang, Q. Zhao, and D. Zheng. Bottlenecks in production networks: An overview. *Journal of Systems Science and Systems Engineering*, 14(3):347–363, 2005.

[35] Y.-C. Wang. Application of reinforcement learning to multi-agent production scheduling. 2003.

[36] T. Yamada and R. Nakano. Job shop scheduling. *IEE control Engineering series*, pages 134–160, 1997.

[37] Z. Zhang, L. Zheng, and M. X. Weng. Dynamic parallel machine scheduling with mean weighted tardiness objective by q-learning. *J. of Advanced Manufacturing Technology*, 34:968–980, 2007.