Dynamic Checkpoint Initiation in Serverless MEC

Saidur Rahman, Apostolos Kalatzis, Mike P. Wittie, David L. Millman, Laura Stanley
Montana State University
Bozeman, MT, USA
{saidur.rahman, apostolos.kalatzis}@student.montana.edu,
{mike.wittie, david.millman, laura.stanley}@montana.edu

2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS) | 978-1-6654-8356-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/COINS54846.2022.9855008

***Abstract*—Mobile applications may want to offload heavy processing jobs to more powerful nodes. Mobile Edge Computing (MEC) can provide low latency and fast processing for the offloaded jobs. Due to the limited capacity of MEC, an efficient checkpointing mechanism can provide fair resource sharing among offloaded jobs by interrupting them as necessary. It is important for an MEC controller to decide when to start checkpointing based on the resource availability and the number of queued requests. In this paper, we propose a resource management framework that decides when to start checkpointing to utilize the MEC compute resources efficiently. We also show how we can integrate the proposed framework in practical implementation of an MEC architecture.**

***Index Terms*—MEC, Serverless, checkpointing**

I. Introduction

The significant growth of mobile technologies brings more complex mobile applications. These applications may need to process rich sensor data on an ongoing basis. The speed of computation in mobile devices is limited by low device processing power and limited battery capacity. Consequently, mobile applications may want to offload computation-intensive data processing jobs onto more powerful nodes, such as cloud servers, to provide faster response times. We define an offloaded job as a long running process supporting multiple low-latency mobile application requests. However, offloading to the cloud brings its own problems. Though the cloud provides greater computational resources, the network latency to the cloud may erode the benefits of faster job execution. So, cloud servers may not help to speed up response time for all latency sensitive offloaded processing jobs. Another computing platform, Mobile Edge Computing (MEC), which is deployed in network proximity to users, is an option to handle latency sensitive processing jobs, offloaded by mobile devices. This deployment improves processing speed, but does not incur as much latency as the cloud. MEC has higher computation capacity than mobile devices, but lower than cloud, and so the opportunity offload onto MEC nodes remains constrained.

Future cellular networks will rely on software-based network traffic processing at Base-band Units (BBUs). BBUs can allow MEC function to use the spare compute resources for on-demand computation service [1]–[3]. However, a BBU will need MEC jobs to free up resources when network traffic surges. Moreover, an MEC node may receive more offloaded jobs than it has the capacity to handle concurrently and queue them for when resources become available. An offloaded job may require computation resources for a long period of time to support multiple low-latency mobile application requests. As a result, the jobs waiting in the queue can experience starvation. So, MEC needs efficient mechanisms to provide fair sharing of computation resources among all jobs. To enforce fair sharing of computation resources, MEC needs to limit the run-time of any offloaded job in each execution period either predictably, or urgently in response spiking BBU demand.

To limit the runtime of an offloaded job, tools such as MicroLambda ($\mu\lambda$) [4] and Checkpoint/Restore In Userspace (CRIU) [5] can suspend a process by checkpointing its state to disk. Consequently, an MEC scheduler can migrate to resume the job to a different MEC node, or wait to resume it on the same node. However, creating and transfering a checkpoint takes time. The checkpointing processing time varies between checkpointing tools, job types, and job execution points. So, MEC needs to know when to start checkpointing before the assigned runtime of a job runs out. Otherwise, the MEC scheduler may start checkpointing early, which will lead to underutilization of runtime, or may fail to finish checkpointing before the runtime ends, which will lead to waste of the assigned computation time. Consequently, it is not clear to the MEC node when to start checkpointing without knowing the duration of checkpointing for the current execution state of each job.

In this work, we present a prediction model, called Penalty-based Multiple Linear Regression (PB-MLR), to estimate job checkpointing duration to decide when to start checkpointing. We develop the model using Multiple Linear Regression (MLR), for which we define a custom cost function. We show how the PB-MLR reduces the number of late checkpoint occurrences and reduces the computation waste resulting from early checkpoints.

We create the model online to learn from every checkpoint creation. We integrate the proposed learning process in the existing $\mu\lambda$ architecture. We show how the proposed online model learns from $\mu\lambda$ program checkpointing state information

978-1-6654-8356-8/22/$31.00 ©2022 IEEE

and corresponding checkpointing duration after checkpointing attempt and reduces the resource waste over the time.

The rest of this paper is organized as follows. Section II provides the background on $\mu\lambda$. Section III presents related work on checkpointing mechanisms and migration. In Section IV, we outline the proposed PB-MLR to predict checkpointing duration and in Section V we present performance evaluation of the PB-MLR. Finally, we conclude in Section VI.

II. Background

To ground further discussion in a concrete system model, we start with background on MicroLambda ($\mu\lambda$) and checkpoint penalty scenarios absent from the original $\mu\lambda$ paper [4].

A. MicroLambda

MicroLambda ($\mu\lambda$) [4], [6] is a checkpointing method to partition offloaded application execution across consecutive runtimes. $\mu\lambda$ dynamically splits stateful and long-running computation across multiple serverless function invocations on edge compute nodes. $\mu\lambda$ dynamically inject checkpointing code at the current line of an offloaded Python process. After the injection, the offloaded process creates a checkpoint and uploads it to the database before terminating. $\mu\lambda$ then restarts the process from the checkpoint at the same, or a different edge node. Compared to the other state of the art checkpointing methods such as Checkpoint/Restore In Userspace (CRIU) [5], $\mu\lambda$ needs less time to create a checkpoint and less memory to store it because $\mu\lambda$ only serializes the internal states of the process.

Figure 1 shows how $\mu\lambda$ manages computation offloading. We will talk about the blue arrows and texts in Section IV-B. In step 1, Client sends an offloading request to the Redis database deployed on edge device such as a BBU node. In step 2, Redis database notifies the manager node to perform the computation job by sending function f. In step 3, the Manager node enrolls one or more worker nodes to install the $\lambda(f)$, a serverless f function if it has not been installed already. After enrolling, the Manager invokes the lambda function on the Worker in step 4. The Worker node runs $\lambda(f)$ until r_{limit}, checkpoints the job, and uploads the state information to Redis shown in step 5. In step 6, Redis notifies the Manager, and the Manager again invokes the lambda function on the same or different Worker node. The Worker node loads the previous state information to resume the execution from the last checkpoint. When the execution of an offloaded job is done that means the entire job is done, the Redis notifies the Client of the finished computation with output o shown in step 7.

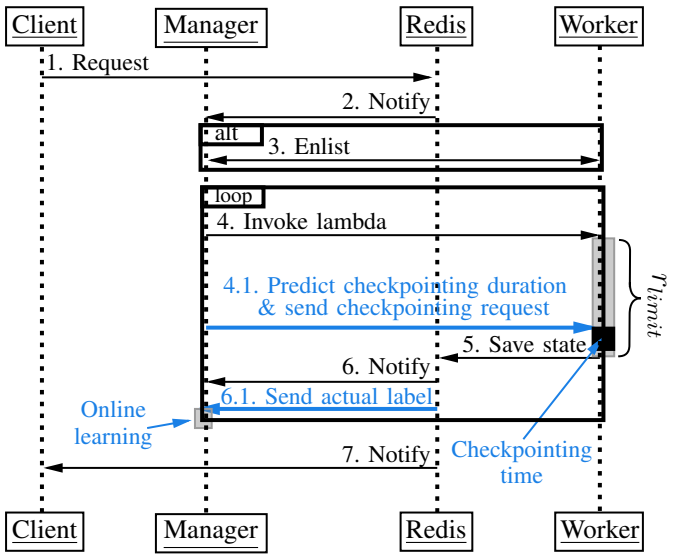
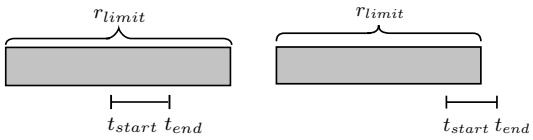


Fig. 1: Messaging and coordination process of $\mu\lambda$ [4] shown in black; updated dynamic checkpoint initiation shown in blue.

B. MicroLambda Penalty Scenarios

Figure 2 shows the early and late checkpoint penalty scenarios, where t_{start} and t_{end} are the start time and end time of the checkpointing. The time to make a checkpoint is $t_{end} - t_{start}$. Figure 2a shows the scenario of early checkpoint where $r_{limit} - t_{end}$ amount of time is resource non-utilization because of early checkpoint. Figure 2b shows the late checkpoint scenario, where the runtime terminates before completing checkpointing. As a result, the task needs to run from the previous checkpoint if there is any; otherwise, it needs to start from the beginning. So, computation resource used by the job for r_{limit} duration will be wasted.



(a) Early checkpoint (b) Late checkpoint

Fig. 2: Checkpointing penalty scenarios.

By default, $\mu\lambda$ checkpoints $\lambda(f)$ at time $r_{limit} - c$ where c is a constant estimate of the checkpointing duration. We will discuss how we find the static checkpointing duration in Section V-A. $\mu\lambda$ assumes c is a value large enough to preclude late checkpoint scenarios. However, c may not eliminate computation wastage. In the evaluation section, we will demonstrate that a static checkpoint time does not eliminate the penalty resulting from early checkpoints, nor does it fully eliminate late checkpoints, because the different execution points of a program may have a different memory footprints, interpreter stack size, and variable state. So, it is important to know when to inject the `save_checkpoint()` function to the Python program, so that the $\mu\lambda$ can utilize the r_{limit} without wasting assigned computation resource.

III. Related Work

$\mu\lambda$ offers low overhead application checkpointing to load balance and migrate jobs between different MEC nodes. Load

balancing is necessary to provide fair resource sharing in the MEC system, because MEC has limited computation resources.

Researchers work on load balancing in many different fields such as operating system [7], [8], cluster system [9], cloud system [10], [11] to provide runtime load balancing and fault tolerance. PB-MLR is related to both load balancing and migration decisions for providing fault tolerant services.

Many methodologies have been proposed to predict future unexpected failures and checkpoints before faults occur. Li et al. presented an adaptive fault management method where they make migration and checkpointing decision based on failure prediction for cluster computing using cost-based evaluation algorithm [12]. Frank et al. proposed an iterative checkpointing algorithm to calculate the interval for jobs to improve the checkpointing intervals rate [13]. In all methods, the authors assume that they will have enough time to complete checkpointing, and that a fault cannot occur in the middle of checkpointing operation. Even in operating systems, the designers developed the checkpointing architectures in such a way that the checkpointing, or interrupting a process to facilitate other process to use the computation resource will be an uninterrupted process. After a successful checkpointing of a process, other process will get the chance to execute [14], [15].

Job migration prediction forcasts when the job needs to transfer. In general, checkpointing a VM, a container, or the OS process does the migration process. Duggan et al. developed a model to predict bandwidth usage and CPU utilization of live migration using Artificial Neural Network [16]. Motaki et al. proposed a model to predict additional CPU usage for VM migration, average power consumption, VM downtime using different regression models such as Ridge Regression, support vector regression, and K-nearest neighbors regression to improve the live migration by reducing violation, occurs when the host cannot obtain the required CPU capacity [17]. Majeed et al. introduced an offloading technique for container migration "Save and Load", where the authors used CRIU [5] for checkpointing and used Multivariate Linear Regression, Polynomial Multivariate, Random forest, and Support Vector Regression to estimate the offloading time for migration after checkpointing [18].

While both VM and container migration are efficient at checkpointing and migrating groups of processes, the overhead of checkpointing the VM or container is significantly higher than that of the process checkpointing in $\mu\lambda$ [4]. Moreover, the methods we discussed above do not predict the checkpointing duration to decide when to start checkpointing within a limited runtime. These predictive methodologies use an offline dataset to develop the model, which is not practical for MEC architecture because the system needs to wait a significant amount of time to build a model, while making uninformed checkpointing decisions in the meantime.

IV. Checkpointing Prediction Framework for MicroLambda ($\mu\lambda$)

We extend the $\mu\lambda$ framework by integrating a checkpointing prediction framework to utilize the r_{limit} efficiently. In this section, we present our proposed model that analyzes a job's behavior, such as memory footprint, interpreter stack size, and current execution time, and predicts the checkpointing duration. The model helps $\mu\lambda$ checkpoint offloaded programs efficiently and reduces penalty discussed in Section II-B.

A. Penalty-based Multiple Linear Regression (PB-MLR)

We use multiple linear regression (MLR) to develop our proposed framework PB-MLR. MLR is a model that assumes that the output variable and each independent variables have linear relationship and uses multiple independent variables to predict the output variable. To create the model, we must fit a set of parameters. One way to fit these parameters is to minimize a cost function using gradient descent [19]. MLR with a standard cost function (Standard-Cost MLR) tries to fit a line to reduce the error between observed and predicted data. The standard cost function of MLR is:

$$cost = \frac{1}{n}\sum_{i=0}^{i=n} d_i^2 \tag{1}$$

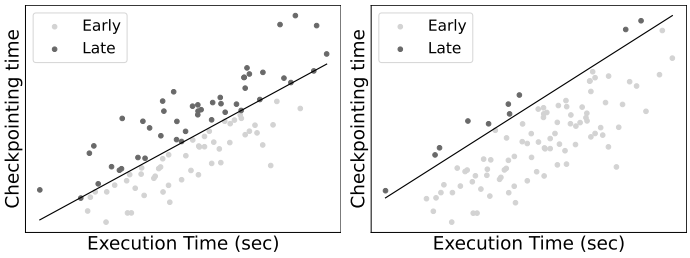
$$d_i = y[i] - \hat{y}[i] \tag{2}$$

where, $y[i]$ and $\hat{y}[i]$ are the observed and predicted time respectively. When $\hat{y}[i]$ is lower than $y[i]$, then late checkpoint occurs. Otherwise, early checkpoint occurs, shown in Figure 2. Figure 3 gives an example of the best fit line of MLR for two different cost scenarios. As shown in Figure 3a, the Standard-Cost MLR equally penalizes early and late checkpoints. However, for $\mu\lambda$, we modified the Equation 2 to reduce the number of late checkpoints and the gap between observed and predicted data as follows. Let $e[i]$ be the current execution time,

$$d_i = \begin{cases} y[i] - \hat{y}[i], & \text{if } \hat{y}[i] \geq y[i] \\ \hat{y}[i] + e[i], & \text{otherwise} \end{cases} \tag{3}$$

When a late checkpoint occurs, $\mu\lambda$ will fail to complete the checkpointing. As a result, we need to redo the computation from the previous checkpoint. So, we set the penalty equal to the total computation time wasted, which is the summation of $e[i]$ and $\hat{y}[i]$. We use the standard cost equation Equation 2 for early checkpoint to reduce the error between predicted and observed data. The proposed cost function tries to reduces the number of late checkpoints and also reduce the distance between predicted and observed values for early checkpoint shown in Figure 3b.

MLR with the proposed penalty-based cost function reduces the number of late checkpoints and also reduces the gap between predicted and observed data for early checkpoint. We choose linear regression instead of the polynomial regression because the linear regression shows a better R^2 score. R^2 score is used to evaluate the performance of regression based machine learning model. For our dataset, we use leave one out cross validation



(a) Standard-Cost MLR (b) Proposed PB-MLR

Fig. 3: Penalty scenarios for Standard-Cost MLR vs. PB-MLR

and linear regression shows R^2 score 0.78 ± 0.14 compared to the n-degree polynomial regression where $n > 1$. The R^2 score of the polynomial regressions are below 0.65.

We use program memory usage, interpreter stack size, current execution time as input features to develop the PB-MLR model. We select the input features by identifying multicollinearity [20]. Multicollinearity occurs when the input features are correlated. We select the input features using p-value and Variance Inflation Factor (VIF) to reduce the multicollinearity. Higher VIF and p-value indicate strong correlation in the input features. The input features we select which are program memory usage, interpreter stack size, current execution time show VIF and p-value are low. That means the input features are not highly correlated. So, we can use these input features to develop PB-MLR model.

B. Online Learning Framework

Our other goal is to integrate PB-MLR with $\mu\lambda$. Initially, the deployed job does not have any labeled information. So, we initialize our proposed model as online where at each checkpoint, we update the model based on observed data. At the beginning, the proposed online model may predict inaccurate checkpointing duration. Over the time, the model will evolve with more data points and will try to reduce the cost based on the proposed cost function.

In Section II-A, we discussed $\mu\lambda$ messaging and coordination process shown in Figure 1. The blue arrows and texts depicts how we integrate the online learning into $\mu\lambda$. After invoking the function $\lambda(f)$ in step 4, the Manager node tracks the jobs and its resource usage to predict checkpointing duration using proposed model. Based on the predicted value from our proposed model, the Manager decides when to start checkpointing and send the checkpointing request to the Worker node shown in step 4.1. The Worker node checkpoints the job using $\mu\lambda$'s dynamic program checkpointing, terminates the job and saves the state to Redis. In step 6, Redis notifies the Manager, which continues the execution by resuming the $\lambda(f)$ from the checkpoint saved in step 5, now with s_1 as the starting state. Redis also notifies the Manager the observed checkpointing duration and the proposed model learns from the new observed data shown in step 6.1.

We develop the data monitoring tool inside of the Manager node using Python libraries: `Py-Spy` [21] and `Memory Profiler` [22]. We use these two libraries to get the information since $\mu\lambda$ uses Python Interpreter Stack size and state of the variables to predict a checkpoint. `Py-Spy` is a sampling profiler to provide information about where a Python program is spending time without modifying the program or restarting the program. It looks at the global `PyInterpreterState` variable of a program to get all the Python threads running in the interpreter and iterates each `PyFrameObject` in each thread to figure out the call stack. We extract the call stack information, including the stack size. `Memory Profiler` [22] is a Python tool to monitor the memory usage of a program. We use time-based memory usage, where we get the memory usage of a program. When the Manager needs to make a checkpoint, the Data Monitoring Tool gathers these data which are used as an input of the proposed model.

V. Evaluation

The goal for our evaluation is to show the practical integration of the PB-MLR with $\mu\lambda$ and how the waste of r_{limit} can be reduced. We want to evaluate the accuracy of $\mu\lambda$ checkpoint prediction mechanism to support different types of edge computing workloads without wasting computation time. We have used the same $\mu\lambda$ architecture [4] and the same three workloads, Face Recognition, Ride hailing and Stress classification representing different classes of computation suitable for offloading to edge computing nodes. $\mu\lambda$ has the facilities to inject the checkpointing code to pause the process when needed. We want to see how PB-MLR works for those applications if we integrate PB-MLR in $\mu\lambda$ architecture.

A. Competing Solutions

We compare the proposed PB-MLR framework's performance to the existing MLR technique, which we are calling Standard-Cost MLR. Standard-Cost MLR uses the Equation 2 cost function to reduce the distance between the predicted and observed data points. We compare the PB-MLR framework with the static value as a checkpointing duration. The previous $\mu\lambda$ uses static checkpointing duration. Though the static value is hard to guess without analyzing the data, we assume a suitable system chooses the best static value as a checkpointing duration. We define such a suitable as a constant value with minimum lost computation by analyzing the offline dataset generated after series of executions.

B. Results

To understand the performance of PB-MLR, we compare it to Standard-Cost MLR and static checkpointing. Specifically, we measure the efficiency of PB-MLR in terms of range of checkpointing duration, percentage of early and late checkpoint occurrence and the total lost computation for different applications.

1) Checkpoint time: Figure 4 shows the evaluated applications on the x-axis and the time in seconds to create a checkpoint on the y-axis. We measure checkpoint time by subtracting the time at which we start the checkpointing process from when a checkpoint is produced on the file system. The light grey boxplots depict the observed checkpointing duration. The grey and black boxplots depict the estimated checkpointing duration of Standard-Cost MLR and the proposed PB-MLR, respectively. Each box plot depicts the distribution of 100 checkpoint durations, taken at random times during application execution. The dotted horizontal line shows a suitable static checkpointing value. We find the suitable static checkpointing by iterating over a fixed set of static values to calculate the total lost computation on every application. The static checkpointing which shows low total lost computation compared to the other static checkpointing values is the suitable checkpointing value.

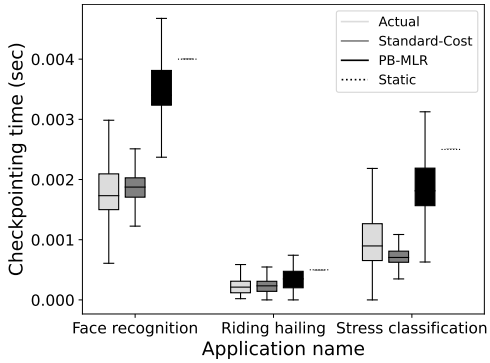


Fig. 4: Checkpointing duration.

In general, we observe that for all applications the prediction range of Standard-Cost MLR is close to the actual checkpointing duration. As a result, there might be a high possibility to start the checkpointing process late since the objective of Standard-Cost MLR is to reduce the residual sum of squares between the observed and targets in the dataset. On the other hand, the proposed PB-MLR predicts a higher value than the actual time to avoid a late checkpoint scenario. The static value is higher than the actual time and also higher than PB-MLR upper quartile.

2) Percentage of early and late checkpoint occurrences: Figure 5 shows the evaluated applications on the x-axis and the percentage of early and late checkpoint prediction on the y-axis. We consider a predicted checkpointing duration early if the predicted value is greater than or equal to the observed checkpointing duration; otherwise, we consider the checkpointing duration as late. The dark grey and light grey with diagonal line pattern show the early and late checkpoint of the PB-MLR in percentage, the solid black and grey with cross pattern series show the early and late checkpoint of the Standard-Cost MLR in percentage, the slate grey and gainsboro grey with dotted pattern show the early and late checkpoint of the static checkpointing respectively. Each stacked bar plot illustrates the percentage of

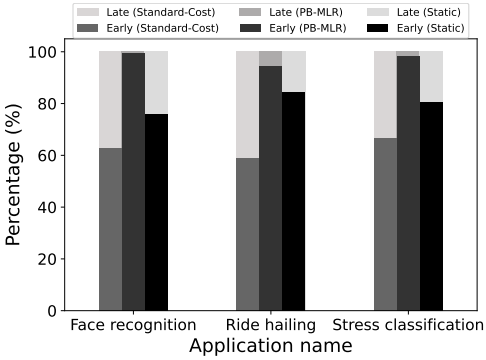


Fig. 5: Percentage of lost computation

predicted early-late checkpoint at random application execution duration to produce checkpoints at different points in application execution. In general, we observe that for all applications, the PB-MLR generates less than 7% late checkpoint compared to Standard-Cost MLR. The percentage of the early checkpoint scenario for face recognition, ride hailing and stress classification applications are 99.46%, 94.5%, and 98.14%, respectively. Though the static checkpointing shows less late checkpoint percentage than Standard-Cost MLR, static checkpointing has higher late checkpointing percentage compared to PB-MLR.

3) Lost Computation: Figure 6 shows the evaluated applications on the x-axis and the total amount of lost computation in seconds because of early and late checkpoint on the y-axis. We calculate the total lost computation using Equation 3. The lost computation for early checkpoint prediction is the difference of predicted and observed time, and the lost computation for late checkpoint prediction is the sum of current execution time from the start and predicted time. The light grey, grey and black bars show the total lost computation for Standard-Cost MLR, Static checkpointing and proposed PB-MLR, respectively. Each bar plot illustrates the distribution of 100 checkpoints made after random application execution duration to produce checkpoints at different points in application execution. For every application, Standard-Cost MLR shows higher total lost computation time compared to the other methods. Though static checkpointing shows better performance than Standard-Cost MLR, it has the higher total lost computation than PB-MLR because the

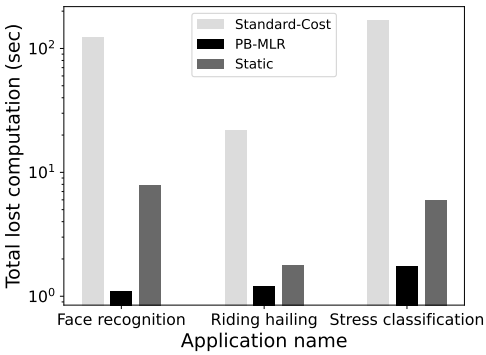


Fig. 6: Total amount of lost computation in seconds

static value does not provide any guarantee to reduce the number of late checkpoint without analyzing the application's current execution state.

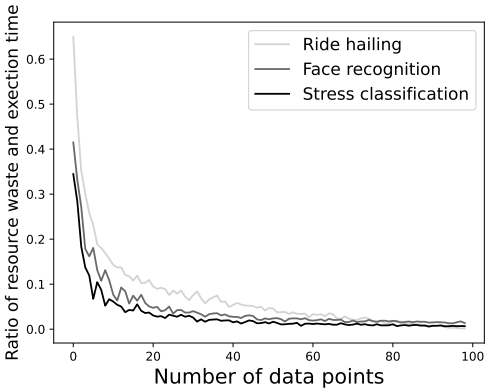


Fig. 7: PB-MLR online learning performance.

4) Online Learning Performance: We worked with the fully trained model from a series of observed data in the previous analysis. Now, we want to see an online learning model using PB-MLR improves with the number of samples. Figure 7 shows on the x-axis number of data points the model learns from, and the y-axis the ratio of resource waste to execution time. We calculate resource waste using the sum of Equation 3 and online learning execution time. Each line plot describes the average of resource waste in 50 trials of data for each x-axis point when checkpointing after a random execution duration. In general, we observe from the graph that the model initially produces inaccurate prediction, which results in higher lost computation higher ratio of resource waste to execution time. Eventually, when the model gets new data and learns from state information and corresponding checkpointing duration, the lost computation declines significantly, which indicates the model produces fewer late checkpoint for all the applications. As every application has a different execution time and learning time, the lost computation varies because the lost computation depends on the current execution time for late checkpoint. Overall, the results show the online learning learns quickly and improves the prediction after learning from only a few checkpoints. It is a practical way to implement on $\mu\lambda$ because initially the newly deployed applications do not have observed information.

the results will improve the performance of $\mu\lambda$ and can provide

VI. Conclusions

In this paper, we showed a framework to predict checkpointing time to limit the runtime of an offloaded job in each execution period. We also illustrated the practical implementation of integrating the proposed framework with the dynamic checkpointing mechanism – MicroLambda ($\mu\lambda$) using online learning. We depicted the results where the proposed framework reduces the waste of MEC compute resources compared to the default $\mu\lambda$ setup. We also show the performance of the online learning setup for integrating existing $\mu\lambda$ architecture. We show fair resource sharing among the processes.

References

[1] M. Barahman, L. M. Correia, and L. S. Ferreira, "A QoS-Demand-Aware Computing Resource Management Scheme in Cloud-RAN," *IEEE Open Journal of the Communications Society*, vol. v1, October 2020.

[2] C.-H. Lin, W.-C. Chien, J.-Y. Chen, C.-F. Lai, and H.-C. Chao, "Energy Efficient Fog RAN (F-RAN) with Flexible BBU Resource Assignment for Latency Aware Mobile Edge Computing (MEC) Services," in *Vehicular Technology Conference (VTC2019-Fall)*, September 2019.

[3] F. Zhang, J. Zheng, Y. Zhang, and L. Chu, "An Efficient and Balanced BBU Computing Resource Allocation Algorithm for Cloud Radio Access Networks," in *Vehicular Technology Conference*, June 2017.

[4] S. Rahman, A. Kalatzis, M. Wittie, A. Elmokashfi, L. Stanley, S. Patterson, and D. L. Millman, "Short and Sweet Checkpoints for C-RAN MEC," in *IEEE Cloud Summit*, October 2021.

[5] "Checkpoint/Restore In Userspace (CRIU)," https://criu.org/, February 2020.

[6] Saidur Rahman, M. P. Wittie, A. Elmokashfi, L. Stanley, and S. Patterson, "MicroLambda - Packetized Computation for 5G Mobile Edge Computing," in *Workshop on Hot Topics in Edge Computing (HotEdge)*, June 2020.

[7] M. Bozyigit and M. Wasiq, "User-level process checkpoint and restore for migration," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 2, April 2001.

[8] J. Walters and V. Chaudhary, "Application-level checkpointing techniques for parallel programs," in *Distributed Computing and Internet Technology*, December 2006.

[9] S. Petri and H. Langendörfer, "Load balancing and fault tolerance in workstation clusters migrating groups of communicating processes," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 4, October 1995.

[10] M. R. Belgaum, S. Soomro, Z. Alansari, and M. Alam, "Cloud service ranking using checkpoint-based load balancing in real-time scheduling of cloud computing," in *Advances in Intelligent Systems and Computing*, February 2018.

[11] D. Singh, J. Singh, and A. Chhabra, "High availability of clouds: Failover strategies for cloud computing using integrated checkpointing algorithms," in *Communication Systems and Network Technologies*, May 2012.

[12] Y. Li and Z. Lan, "Exploit failure prediction for adaptive fault-tolerance in cluster computing," in *CCGRID*, May 2006.

[13] A. Frank, M. Baumgartner, R. Salkhordeh, and A. Brinkmann, "Improving checkpointing intervals by considering individual job failure probabilities," in *Parallel and Distributed Processing Symposium*, May 2021.

[14] A. Zarrabi, "Dynamic Transparent General Purpose Process Migration For Linux," *Grid Computing and Applications (IJGCA)*, vol. 3, no. 4, December 2013.

[15] C. D. V. S. Rao, D. M. E. Naidu, K. Subbaiah, and N. H. R. Reddy, "Process Migration in Network of Linux System," *Computer Science and Network Securit (IJCSNS)*, vol. 7, no. 5, May 2007.

[16] M. Duggan, R. Shaw, J. Duggan, E. Howley, and E. Barrett, "A multitime-steps-ahead prediction approach for scheduling live migration in cloud data centers," *Software: Practice and Experience*, vol. 49, no. 4, September 2018.

[17] S. E. Motaki, A. Yahyaouy, and H. Gualous, "A prediction-based model for virtual machine live migration monitoring in a cloud datacenter," *Computing*, August 2021.

[18] A. Abdul Majeed, P. Kilpatrick, I. Spence, and B. Varghese, "Performance Estimation of Container-Based Cloud-to-Fog Offloading," in *Utility and Cloud Computing Companion (UCC)*, December 2019.

[19] T. Hu, Q. Wu, and D.-X. Zhou, "Convergence of gradient descent for minimum error entropy principle in linear regression," *IEEE Transactions on Signal Processing*, vol. 64, no. 24, September 2016.

[20] Jamal I. Daoud, "Multicollinearity and Regression Analysis," *Journal of Physics: Conference Series*, vol. 949, December 2017.

[21] B. Frederickson, "Py-Spy," https://github.com/benfred/py-spy/, August 2018.

[22] F. Pedregosa and P. Gervais, "Memory Profiler," https://github.com/pythonprofilers/memory_profiler/, December 2021.